CÓDIGO EN LÍNEA ¡GRATIS!

mcgraw-hill-educacion.com



C++

Soluciones de programación

- Aprenda técnicas poderosas de programación en C++
- Maximice el código en línea y las instrucciones paso a paso
- Logre resultados en menos tiempo



Herb Schildt

C++
Soluciones de programación

Acerca del autor

Herbert Schildt es una de las principales autoridades en C++, C, Java y C# y es maestro programador en Windows. Se han vendido más de 3.5 millones de copias de los libros sobre programación de Herb en todo el mundo y se han traducido a todos los idiomas importantes. Es autor de gran cantidad de bestsellers de C++, incluidos *C++: The Complete Reference*, *C++: A Beginner's Guide*, *C++ from the Ground Up* y *STL Programming form the Ground Up*. Sus otros best sellers incluyen *C: Manual de referencia; Java: Manual de referencia; Fundamentos de Java; Java, soluciones de programación y Java 2: Manual de referencia.* Schildt tiene títulos de grado y posgrado de la Universidad de Illinois. Su sitio Web es **www.HerbSchildt.com**.

Acerca del editor técnico

Jim Keogh introdujo la programación en PC en Estados Unidos en su columna *Popular Electronics Magazine* en 1982, cuatro años después de que Apple Computer empezó en una cochera.

Fue integrante del equipo que construyó una de las primeras aplicaciones de Windows para una firma de Wall Street, presentada por Bill Gates en 1986. Keogh ha dedicado casi dos décadas a desarrollar sistemas de cómputo para firmas de Wall Street, como Salomon, Inc., y Bear Stearns, Inc.

Keogh formó parte del cuerpo docente de la Universidad de Columbia, donde impartió cursos de tecnología, incluido el laboratorio de desarrollo de Java. Desarrolló y dirigió la carrera de comercio electrónico en la Universidad de Columbia. Actualmente es parte del cuerpo docente de la Universidad de Nueva York. Es autor de *J2EE: The Complete Reference*, *J2ME: The Complete Reference*, ambos publicados por McGraw-Hill, y más de 55 títulos adicionales. Entre sus otros libros se incluyen *Linux Programming for Dummies*, *Unix Programming for Dummies*, *Java Database Programming for Dummies*, *Essential Guide to Networking*, *Essential Guide to Computer Hardware*, *The C++ Programmer's Notebook* y *E-Mergers*.

C++
Soluciones de programación

Herb Schildt

Traducción

Eloy Pineda Rojas
Traductor profesional



MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • MADRID • NUEVA YORK
SAN JUAN • SANTIAGO • SÃO PAULO • AUCKLAND • LONDRES • MILÁN • MONTREAL
NUEVA DELHI • SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TORONTO

Director editorial: Fernando Castellanos Rodríguez
Editor de desarrollo: Miguel Ángel Luna Ponce
Supervisor de producción: Marco Antonio Gómez Ortiz

C++ SOLUCIONES DE PROGRAMACIÓN



Corporativo Punta Santa Fe
Prolongación Paseo de la Reforma 1015, Torre A,
Piso 17, Colonia Desarrollo Santa Fe,
Delegación Álvaro Obregón,
C.P. 01376, México, D.F.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN: 978-970-10-7266-0

Traducido de la primera edición de
Herb Schildt's *C++ Programming Cookbook*
By: Herb Schildt

ISBN: 978-0-07-148860-0

1234567890 0876543219

Impreso en México *Printed in Mexico*

The McGraw-Hill Companies

Contenido

Introducción

Con los años, amigos y lectores pidieron un libro de soluciones para Java, donde compartiera algunas de las técnicas y los métodos que uso cuando programo. Desde el principio me gustó la idea, pero no lograba darme tiempo para ella en un calendario de escritura muy ocupado. Como muchos lectores saben, escribo demasiado acerca de varias facetas de la programación, con énfasis especial en C++, Java y C#. Debido a los rápidos ciclos de revisión de estos lenguajes, dedico casi todo mi tiempo disponible a actualizar mis libros para que cubran las versiones más recientes de esos lenguajes. Por fortuna, a principios de 2007 se abrió una ventana de oportunidad y finalmente pude dedicar tiempo al proyecto. Empecé con Java, lo que llevó a mi primer *Soluciones de programación de Java*. En cuanto terminé el libro de Java, pasé a C++. El resultado es, por supuesto, este libro. Debo admitir que ambos proyectos están entre los que más he disfrutado.

Con base en el formato de soluciones, este libro destila la esencia de muchas técnicas de propósito general en un conjunto de técnicas paso a paso. En cada una se describe un conjunto de componentes clave, como clases, funciones y encabezados. Luego se muestran los pasos necesarios para ensamblar esos componentes en una secuencia de código que logre los resultados deseados. Esta organización facilita la búsqueda de técnicas en que está interesado para ponerla *en acción*.

En realidad, "en acción" es una parte importante de este libro. Creo que los buenos libros de programación contienen dos elementos: teoría sólida y aplicación práctica. En las soluciones, las instrucciones paso a paso y los análisis proporcionan la teoría. Para llevar esa teoría a la práctica, siempre se incluye un ejemplo completo de código. En los ejemplos se demuestra en forma concreta, sin ambigüedades, la manera en que pueden aplicarse. En otras palabras, en los ejemplos se eliminan las "adivinanzas" y se ahorra tiempo.

Aunque ningún libro puede incluir todas las soluciones que pudieran desearse (hay un número casi ilimitado de ellas), traté de abarcar un amplio rango de temas. Mis criterios para incluir una solución se analizan de manera detallada en el capítulo 1, pero, en resumen, incluí las que serían útiles para muchos programadores y que responderían preguntas frecuentes. Aun con estos criterios, fue difícil decidir qué incluir y qué dejar fuera. Ésta fue la parte más desafiante de la escritura del libro. Al final, se impusieron la experiencia, el juicio y la intuición. Por fortuna, ¡he incluido algo para satisfacer a cada programador!

HS

Código de ejemplo en Web

El código fuente para todos los ejemplos de este libro está disponible de manera gratuita en Web en **www.mcgraw-hill-educacion.com**

Más de Herbert Schildt

C++ Soluciones de programación es sólo uno de los muchos libros de programación de Herb. He aquí algunos otros que le resultarán de interés:

Para aprender más acerca de C++, estos libros le resultarán especialmente útiles.

C++: The Complete Reference

C++: A Begginer's Guide

C++ from the Ground Up

STL Programming from the Ground Up

The Art of C++

Para aprender más acerca de Java recomendamos:

Java Soluciones de programación

Java: Manual de referencia, séptima edición

Java 2: Manual de referencia

Fundamentos de Java

Swing: A Begginer's Guide

Para aprender acerca de C#, sugerimos los siguientes libros de Schildt:

C#: The Complete Reference

C#: A Begginer's Guide

Si quiere aprender acerca del lenguaje C, entonces le interesará el siguiente título.

C: Manual de referencia

Cuando necesite respuestas sólidas, rápidas, busque algo de Herbert Schildt, la autoridad reconocida en programación.

1

CAPÍTULO

Revisión general

En este libro se presenta una colección de técnicas que muestran la manera de realizar varias tareas de programación en C++. En él, se usa el formato de "soluciones". Con cada una se ilustra la manera de realizar una operación específica. Por ejemplo, hay soluciones que leen bytes de un archivo, invierten una cadena, ordenan el contenido de un contenedor, forman datos numéricos, etc. De la misma manera que una receta en un libro de cocina describe un conjunto de ingredientes y una secuencia de instrucciones necesarias para preparar un platillo, cada técnica de este libro describe un conjunto de elementos clave de un programa y la secuencia de pasos necesarios que debe usarse para completar una tarea de programación.

Al final de cuentas, el objetivo de este libro es ahorrar tiempo y esfuerzo durante el desarrollo de un programa. Muchas tareas de programación constan de un conjunto estándar de funciones y clases, que debe aplicarse en una secuencia específica. El problema es que en ocasiones no sabe cuáles funciones usar o qué clases son apropiadas. En lugar de tener que abrirse paso entre grandes cantidades de documentación y tutoriales en línea para determinar la manera de encarar alguna tarea, puede buscar su solución. En cada solución se muestra una manera de llegar a una secuencia, describiendo los elementos necesarios y el orden en que deben usarse. Con esta información, puede diseñar una solución que se amolde a su necesidad específica.

Qué contiene

Este libro no es exhaustivo. El autor decidió qué incluir y dejar fuera. Al elegir las soluciones para este libro, el autor se concentró en cuatro áreas principales: manejo de cadenas, biblioteca estándar de plantillas (STL, Standard Template Library), E/S y formato de datos. Se trata de temas esenciales que interesan a una amplia variedad de programadores. Son temas muy extensos, que requieren muchas páginas para explorarse a fondo. Como resultado, cada uno de estos temas se volvió la base para uno o más capítulos. Sin embargo, es importante establecer que el contenido de esos capítulos no está limitado sólo a esos temas. Como la mayoría de los lectores sabe, casi todo en C++ está interrelacionado. En el proceso de crear soluciones para un aspecto de C++, suelen incluirse varios otros, como localización, asignación dinámica, o sobrecarga de operadores. Por tanto, también suelen ilustrar otras técnicas de C++.

Además de las soluciones relacionadas con los temas principales, se añadieron otras que el autor deseaba incluir pero que no abarcarían un capítulo completo. Éstas se agruparon en el capítulo final. Varias de esas soluciones se concentran en la sobrecarga de operadores más especializados de C++, como **[]**, –>, **new** y **delete**. Otras ilustran el uso de las clases **auto_ptr** y **complex** o muestran cómo crear una función de conversión, un constructor de copia o uno explícito. También hay una solución que demuestra el ID de tipo en tiempo de ejecución.

Por supuesto, la elección de los temas sólo fue el principio del proceso de selección. Dentro de cada categoría, se tuvo que decidir qué incluir y qué dejar fuera. En general, se incluyó una solución si cumple los dos criterios siguientes:

1. La técnica es útil para un amplio rango de programadores.
2. Proporciona una respuesta a una pregunta frecuente de programación.

El primer criterio se explica por sí solo. Se incluyeron soluciones que describen la manera de completar un conjunto de tareas que, por lo general, se encontrarían cuando se crean aplicaciones de C++. Algunas de ellas ilustran un concepto general que puede adaptarse para resolver varios tipos diferentes de problemas. Por ejemplo, en el capítulo 2 se muestra una solución que busca una sustitución dentro de una cadena. Este procedimiento general es útil en varios contextos, como encontrar una dirección de correo electrónico o un número telefónico dentro de una frase, o extraer una palabra clave de una consulta de base de datos. Otras soluciones describen técnicas más específicas pero usadas ampliamente. Por ejemplo, en el capítulo 6 se muestra cómo formar la fecha y la hora.

El segundo criterio se basa en la experiencia del autor en libros de programación. Durante los años en que ha estado escribiendo, le han planteado cientos y cientos de preguntas tipo "¿Cómo hacer?" por parte de los lectores. Estas preguntas vienen de todas las áreas de programación de C++ y van de muy fáciles a muy difíciles. Sin embargo, ha encontrado que un núcleo central de preguntas se presenta una y otra vez. He aquí un ejemplo: "¿Cómo formo un número para que tenga dos lugares decimales?" He aquí otra: "¿Cómo creo un objeto de función?" Hay muchas otras. Estos mismos tipos de preguntas también se presentan con frecuencia en varios foros de programadores en Web. El autor utiliza estas preguntas frecuentes para guiar su selección.

Las soluciones de este libro abarcan varios niveles de habilidad. Algunas ilustran técnicas básicas, como leer bytes de un archivo o sobrecargar el operador **<<** para dar salida a objetos de una clase personalizada. Otras son más avanzadas, como usar la biblioteca de localización para formar valores monetarios, convertir una cadena en fichas o sobrecargar el operador **[]**. Por tanto, el nivel de dificultad de una solución individual puede ir de relativamente fácil a muy avanzado. Por supuesto, casi todo en programación es fácil una vez que sabe cómo hacerlo, pero difícil cuando no. Por tanto, no se sorprenda si algunas parecen obvias. Sólo significa que sabe cómo realizar esa tarea.

Cómo están organizadas

Cada solución de este libro usa el mismo formato, que tiene las siguientes partes:

- Una tabla de elementos clave usados por la solución.
- Una descripción del problema que resuelve.
- Los pasos necesarios para completarla.
- Un análisis a profundidad de los pasos.

- Un ejemplo de código que aplica la solución.
- Opciones que sugieren otras maneras de llegar a una solución.

Una solución empieza por describir la tarea que se realizará. Los elementos clave empleados se muestran en una tabla. Entre éstas se incluyen funciones, clases y encabezados necesarios. Por supuesto, llevar una solución a la práctica puede implicar el uso de elementos adicionales, pero los elementos clave son fundamentales para la tarea que se tiene a mano.

Cada solución presenta entonces instrucciones paso a paso que resumen el procedimiento. A éstas les sigue un análisis a fondo de los pasos. En muchos casos, el resumen bastará, pero los detalles estarán allí si los necesita.

A continuación, se presenta un ejemplo de código que muestra la solución ejecutándose. Todos los ejemplos de código se presentan completos. Esto evita ambigüedades y le permite ver con claridad precisamente lo que está sucediendo sin tener que llenar detalles adicionales. En ocasiones, se incluye un ejemplo extra que ilustra aún más la manera en que puede aplicarse la solución.

Se concluye con un análisis de varias opciones. Esta sección es especialmente importante porque sugiere diferentes modos de implementar una solución u otra manera de pensar en el problema.

Una breve advertencia

Cuando utilice este libro debe tener en cuenta algunos elementos importantes. En primer lugar, una solución muestra *una manera* de resolver una situación. Es posible que existan (y a menudo existen) otras maneras. Tal vez su aplicación específica requiera un método diferente del mostrado. Las soluciones de este libro pueden servir como puntos de partida, ayudar a elegir un método general para llegar a una respuesta y despertar su imaginación. Sin embargo, en todos los casos, debe determinar lo que es apropiado para su aplicación, y lo que no lo es.

En segundo lugar, es importante entender que los ejemplos de código *no* están optimizados para su desempeño. *Están optimizados para clarificar y mejorar la comprensión.* Su propósito es ilustrar con claridad los pasos de la solución. En muchos casos, tendrá pocos problemas al escribir un código más eficiente o corto. Además, los ejemplos son exactamente eso: ejemplos. Son usos simples que no necesariamente reflejan el modo en que escribirá el código para su propia aplicación. En todas las circunstancias, debe crear su propio método que se adapte a las necesidades de su aplicación.

En tercer lugar, cada ejemplo de código contiene el manejo de errores apropiado para ese ejemplo específico, pero tal vez no sea idóneo en otras situaciones. En todos los casos, debe manejar apropiadamente los diversos errores y excepciones que pueden resultar cuando adapte un procedimiento para usarlo en su propio código. Es necesario repetir esto de otra manera. Cuando se implementa una solución, debe proporcionar el manejo de errores apropiado para su aplicación. No basta simplemente con suponer que la manera en que se manejan (o se dejan de manejar) los errores o excepciones en un ejemplo es suficiente o adecuada para su uso. Por lo general, se requerirá manejo adicional de errores en las aplicaciones reales.

Es necesaria experiencia en C++

Este libro es para todos los programadores en C++, sean principiantes o experimentados. Sin embargo, en él se supone que el lector cuenta con los fundamentos de la programación en C++,

incluidas las palabras clave y la sintaxis, y que está familiarizado con las funciones y las clases centrales de las bibliotecas. También debe tener la capacidad de crear, compilar y ejecutar programas de C++. Nada de esto se enseña aquí. (En este libro sólo se trata la aplicación de C++ a diversos problemas de programación. No intenta enseñar fundamentos del lenguaje C++.) Si necesita mejorar sus habilidades en C++, se recomiendan los libros *C++: The Complete Reference*, *C++ From the Ground Up* y *C++: A Beginner's Guide*, de Herb Schildt. Publicados por McGraw-Hill, Inc.

¿Qué versión de C++?

El código y los análisis de este libro se basan en el estándar internacional ANSI/ISO para C++. A menos que se determine explícitamente, no se usan extensiones que no son estándar. Como resultado, casi todas las técnicas presentadas aquí son transportables y pueden usarse con cualquier compilador de C++ que se adhiera al estándar internacional para C++. El código de este libro se desarrolló con Visual C++ de Microsoft. Se usaron tanto Visual Studio como Visual C++ Express (que está disponible sin costo alguno en Microsoft).

NOTA *Al momento de escribir este libro, el estándar internacional para C++ está en proceso de actualización. Se están contemplando muchas características nuevas. Sin embargo, ninguna de ellas aún es parte de C++, ni se usa en este libro. Por supuesto, en futuras ediciones de este libro se utilizarán estas nuevas características.*

Dos convenciones de codificación

Antes de pasar a las soluciones, hay dos temas que deben atenderse y que se relacionan con la manera en que está escrito el código de este libro. El primero se relaciona con el regreso de un valor desde **main()**. El segundo se relaciona con el uso de **namespace std**. A continuación se explican las decisiones tomadas en relación con estas dos características.

Regreso de un valor de main()

Los ejemplos de código de este libro siempre devuelven explícitamente un valor entero de **main()**. Por convención, un valor devuelto de cero indica una terminación exitosa. Un valor diferente de cero indica alguna forma de error.

Sin embargo, no es necesaria la devolución explícita de un valor de **main()**, porque, en palabras del estándar internacional para C++:

"Si el control alcanza el final de `main` sin encontrar una instrucción `return`, el efecto es ejecutar `return 0;`"

Por esto, en ocasiones encontrará código que no devuelve explícitamente un valor de **main()**, dependiendo en cambio del valor de devolución implícito de cero. Pero éste *no es* el método usado en este libro.

En cambio, todas las funciones de **main()** en este libro devuelven explícitamente un valor, por dos razones. En primer lugar, algunos compiladores lanzan una advertencia cuando un método diferente de **void** no regresa un valor de manera explícita. Para evitar esta advertencia, **main()** debe incluir una instrucción **return**. En segundo lugar, ¡parece una buena práctica devolver explícitamente un valor, puesto que **main()** está declarado con un tipo de devolución **int**!

¿Uso del espacio de nombres std?

Uno de los problemas que encara el autor de un libro de C++ es si se usa o no la línea:

```
using namespace std;
```

casi en la parte superior de cada programa. Esta instrucción trae a la vista el contenido del espacio de nombres **std**. Éste contiene la biblioteca estándar de C++. Por tanto, al usar el espacio de nombres **std**, se trae la biblioteca estándar al espacio de nombres global, y es posible hacer referencia directa a nombres como **cout**, en lugar de **std::cout**.

El uso de

```
using namespace std;
```

es muy común y, en ocasiones, polémico. A algunos programadores les desagrada, lo que sugiere que abona en contra del empaquetamiento de la biblioteca estándar en el espacio de nombres **std** y atrae conflictos con código de terceros, sobre todo en proyectos grandes. Aunque esto es cierto, otros señalan que en programas cortos (como los ejemplos mostrados en este libro) y en proyectos pequeños, la conveniencia que ofrece supera fácilmente la posibilidad remota de conflictos, lo que rara vez ocurre (si llega a suceder) en estos casos. Francamente, en programas para los que el riesgo de conflictos es, en esencia, nulo, tener que escribir siempre **std::cout**, **std::cin**, **std::ofstream**, **std::string**, etc., *es* tedioso. También hace el código más extenso.

Mientras el debate continúa, en este libro se usa

```
using namespace std;
```

en los programas de ejemplo, por dos razones. En primer lugar, acorta el código, lo que significa que puede caber más código en una línea. En un libro, la longitud de una línea está limitada. Al no tener que usar constantemente **std::** se acortan las líneas, lo que significa que cabrá más código en una línea sin que ésta se tenga que dividir. Cuanto menor sea la cantidad de líneas divididas, más fácil será leer el código. En segundo lugar, hace que los ejemplos de código sean menos extensos, lo que mejora su claridad en la página impresa. De acuerdo con la experiencia del autor, **using namespace std** es muy útil cuando se muestran en un libro los programas de ejemplo. Sin embargo, su uso en los ejemplos *no* significa el respaldo de la técnica, en general. El lector debe decidir lo apropiado para sus propios programas.

2

CAPÍTULO

Manejo de cadenas

Casi siempre hay más de una manera de hacer algo en C++. Ésta es una razón por la que C++ es un lenguaje tan rico y poderoso. Le permite al programador elegir el mejor método para la tarea a mano. En ningún lado es más evidente este aspecto de varias facetas de C++ que en las cadenas. En C++, las cadenas se basan en dos subsistemas separados pero interrelacionados. Un tipo de cadena se hereda de C. El otro está definido en C++. Juntos, proporcionan al programador dos maneras diferentes de pensar y manejar secuencias de caracteres.

El primer tipo de cadena al que da soporte C++ es la *cadena terminada en un carácter nulo*. Se trata de la matriz **char** que contiene los caracteres que componen una cadena, seguida por null. La cadena terminada en un carácter nulo se hereda de C y le da un control de bajo nivel sobre operaciones de cadena. Como resultado, la cadena terminada en un carácter nulo ofrece una manera muy eficiente de manejar las secuencias de caracteres. C++ también da soporte a cadenas de caracteres amplias, terminadas en un carácter nulo, que son matrices de tipo **wchar_t**.

El segundo tipo de cadena es un objeto de tipo **basic_string**, que es una clase de plantilla definida por C++. Por tanto, **basic_string** define un tipo único cuyo propósito es representar secuencias de caracteres. Debido a que define un tipo de clase, ofrece un método de alto nivel para trabajar con cadenas. Por ejemplo, define muchas funciones de miembros que realizan varias manipulaciones de cadenas, y varios operadores de sobrecarga para operaciones de cadena. Hay dos especializaciones de **basic_string** que están definidas por C++: **string** y **wstring**. La clase **string** opera en caracteres de tipo **char**, y **wstring** opera en caracteres de tipo **wchar_t**. Por tanto, **wstring** encapsula una cadena de caracteres ampliados.

Como se acaba de explicar, las cadenas terminadas en un carácter nulo y **basic_string** soportan cadenas de tipo **char** y **wchar_t**. La principal diferencia entre cadenas basadas en **char** y en **wchar_t** es el tamaño del carácter. De otro modo, los dos tipos de cadenas se manejan, en esencia, de la misma manera. Por conveniencia y debido a que las cadenas basadas en **char** son, por mucho, las más comunes, constituyen el tipo de cadenas utilizadas en las soluciones de este capítulo. Sin embargo, con poco esfuerzo pueden adoptarse las mismas técnicas básicas para cadenas de carácter ampliado.

El tema de las cadenas en C++ es muy extenso. Francamente, sería fácil llenar un libro completo con código relacionado con ellas. Por tanto, limitar las soluciones de cadenas a un solo capítulo representa todo un desafío. Al final, se seleccionaron las que responden preguntas comunes, ilustran aspectos clave de cada tipo de cadena o demuestran principios generales que pueden adaptarse a una amplia variedad de usos.

He aquí las soluciones contenidas en este capítulo:

- Realice operaciones básicas en cadenas terminadas en un carácter nulo
- Busque una cadena terminada en un carácter nulo
- Invierta una cadena terminada en un carácter nulo
- Ignore diferencias entre mayúsculas y minúsculas cuando compare cadenas terminadas en un carácter nulo
- Cree una función de búsqueda y reemplazo para cadenas terminadas en un carácter nulo
- Ordene en categorías caracteres dentro de una cadena terminada en un carácter nulo
- Convierta en fichas una cadena terminada en un carácter nulo
- Realice operaciones básicas en objetos de **string**
- Busque un objeto **string**
- Cree una función de búsqueda y reemplazo para objetos **string**
- Opere en objetos **string** mediante iteradores
- Cree una búsqueda no sensible a mayúsculas y minúsculas y funciones de búsqueda y reemplazo para objetos **string**
- Convierta un objeto **string** en una cadena terminada en un carácter nulo
- Implemente la resta para objetos **string**

NOTA *Una cobertura a fondo de las cadena terminada en un carácter nulo y la clase* ***string*** *se encuentra en el libro* C++: The Complete Reference*, de Herb Schildt.*

Revisión general de las cadenas terminadas en un carácter nulo

El tipo de cadena más común empleado en un programa C++ es la cadena terminada en un carácter nulo. Como se mencionó, se trata de una matriz de **char** que termina con un carácter nulo. Por tanto, una cadena terminada en un carácter nulo *no* es, en sí, un tipo único. En cambio, es una *convención* reconocida por todos los programadores en C++. La cadena terminada en un carácter nulo está definida en el lenguaje C y la mayoría de los programadores en C++ aún la usan ampliamente. También se hace referencia a ella como una *cadena* ***char**** o, en ocasiones, como una *cadena C*. Aunque las cadenas terminadas en un carácter nulo son un territorio familiar para la mayoría de los programadores en C++, aún es útil revisar sus atributos y capacidades clave.

Hay dos razones por las que las cadenas terminadas en un carácter nulo se usan ampliamente en C++. En primer lugar, todas las literales de cadena están representadas como cadenas terminadas en un carácter nulo. Por tanto, cada vez que crea una literal de cadena, está creando una cadena terminada en un carácter nulo. Por ejemplo, en la instrucción

```
const char *ptr = "Hola";
```

la literal "Hola" es una cadena terminada en un carácter nulo. Esto significa que es una matriz **char** que contiene los caracteres **Hola** y termina en un valor nulo. En esta instrucción, un apuntador a la matriz se asigna a **ptr**. Resulta interesante observar que **ptr** se especifica como **const**. El estándar de C++ especifica que las literales de cadena son matrices de tipo **const char**. Por tanto, es mejor usar un apuntador **const char *** para apuntar a una. Sin embargo, el estándar actual también define una conversión automática (pero ya desautorizada) a **char ***, y es muy común ver código en que se omite **const**.

La segunda razón por la que las cadenas terminadas en un carácter nulo se usan ampliamente es la eficiencia. El empleo de una matriz terminada en un carácter nulo para contener una cadena permite la implementación de operaciones con muchas cadenas de una manera muy fina. (En esencia, las operaciones con este tipo de cadenas son simplemente operaciones especializadas con matrices.) Por ejemplo, he aquí una manera de escribir la función de la biblioteca estándar **strcpy()**, que copia el contenido de una cadena en otra.

```
// Una manera de implementar la función strcpy() estándar.
char *strcpy(char *destino, const char *origen) {
  char *d = destino;
   // Copia el contenido del origen en el destino.
  while(*origen) *destino++ = *origen++;
   // El destino termina en un carácter nulo.
  *destino = '\0';
   // Devuelve el apuntador al principio del destino.
  return d;
}
```

Preste especial atención a la línea:

```
while(*origen) *destino++ = *origen++;
```

Debido a que la cadena de origen termina con un carácter nulo, puede crearse un bucle muy eficiente que simplemente copia caracteres hasta que el carácter al que señala **destino** es nulo. Recuerde que en C++ cualquier valor diferente de cero es verdadero, pero cero es falso. Debido a que el carácter nulo es cero, el bucle **while** se detiene cuando se encuentra el terminador nulo. Bucles como el que se acaba de mostrar son comunes cuando se trabaja con cadenas terminadas en un carácter nulo.

La biblioteca C++ estándar define varias funciones que operan en cadenas terminadas en un carácter nulo. Esto requiere el encabezado **<cstring>**. Estas funciones serán familiares, sin duda, para muchos lectores. Más aún, las soluciones en este capítulo explican por completo las funciones de cadena que emplean. Sin embargo, aún es útil presentar una breve lista de las funciones más comunes de cadenas terminadas en un carácter nulo.

Función	**Descripción**
char *strcat(char **cad1*, const char **cad2*)	Une la cadena señalada por *cad2* al final de la cadena señalada por *cad1*. Devuelve *cad1*. Si la cadena se superpone, el comportamiento de **strcat()** queda indefinido.
char *strchr(const char **cad*, int *car*)	Devuelve un apuntador a la primera aparición del byte de orden bajo de *car* en la cadena a la que señala *cad*. Si no se encuentran coincidencias, se devuelve un apuntador nulo.
int strcmp(const char **cad1*, const char *cad2*)	Compara lexicográficamente la cadena señalada por *cad1* con la señalada por *cad2*. Devuelve menos de cero si *cad1* es menor que *cad2*, más de cero si *cad1* es mayor que *cad2* y cero si las cadenas son iguales.

Función	Descripción
char *strcpy(char **destino*, const char **origen*)	Copia la cadena señalada por *origen* en la cadena señalada por *destino*. Regresa *destino*. Si la cadena se superpone, el comportamiento de **strcpy()** queda indefinido.
size_t strcspn(const char **cad1*, const char **cad2*)	Devuelve el índice del primer carácter en la cadena señalada por *cad1* que coincide con cualquier carácter en la cadena apuntada por *cad2*. Si no se encuentra una coincidencia, se devuelve la longitud de *cad1*.
size_t strlen(const char **cad*)	Devuelve el número de caracteres en la cadena señalada por *cad*. No se cuenta el terminador nulo.
char *strncat(char **cad1*, const char **cad2*, size_t *cuenta*)	Une no más de *cuenta* caracteres de la cadena señalada por *cad2* al final de *cad1*. Devuelve *cad1*. Si las cadenas se superponen, el comportamiento de **strncat()** queda indefinido.
char *strncmp(const char **cad1*, const char **cad2*, size_t *cuenta*)	Compara lexicográficamente no más de los primeros *cuenta* caracteres en la cadena señalada por *cad1* con la señalada por *cad2*. Devuelve menos de cero si *cad1* es menor que *cad2*, más de cero si *cad1* es mayor que *cad2* y cero si las cadenas son iguales.
char *strncpy(char **destino*, const char **origen*, size_t *cuenta*)	Copia no más de *cuenta* caracteres de la cadena señalada por *origen* en la cadena señalada por *destino*. Si *origen* contiene menos de *cuenta* caracteres, los caracteres nulos se añadirán al final de *destino* hasta que *cuenta* caracteres se hayan copiado. Sin embargo, si *origen* es mayor que *cuenta* caracteres, la cadena resultante ya no terminará en un carácter nulo. Devuelve *destino*. Si la cadena se superpone, el comportamiento de **strcpy()** queda indefinido.
char *strpbrk(const char **cad1*, const char **cad2*)	Devuelve un apuntador al primer carácter de la cadena señalada por *cad1* que coincide con cualquier carácter de la cadena señalada por *cad2*. Si no se encuentra una coincidencia, se devuelve un apuntador nulo.
char *strrchr(const char **cad*, int car)	Devuelve un apuntador a la última aparición del byte de orden bajo de *car* en la cadena señalada por *cad*. Si no se encuentra una coincidencia, se devuelve un apuntador nulo.
size_t strspn(const char **cad1*, const char **cad2*)	Devuelve el índice del primer carácter en la cadena señalada por *cad1* que no coincide con cualquier carácter en la cadena apuntada por *cad2*.
char *strstr(const char **cad*1, const char **cad*2)	Devuelve un apuntador a la primera aparición de la cadena señalada por *cad2* en la cadena señalada por *cad1*. Si no se encuentra una coincidencia, se devuelve un apuntador nulo.
char *strtok(char **cad*, const char **delims*)	Devuelve un apuntador a la siguiente ficha en la cadena señalada por *cad*. Los caracteres de la cadena señalada por *delims* especifican los delimitadores que determinan los límites de una ficha. Se devuelve un apuntador nulo cuando no hay una ficha que devolver. Para convertir una cadena en ficha, la primera llamada a **strtok()** debe hacer que *cad* señale a la cadena que se convertirá en ficha. Llamadas posteriores deben pasar un apuntador nulo a *cad*.

Observe que varias de las funciones, como **strlen()** y **strspn()**, usan el tipo **size_t**. Se trata de una forma de entero no asignado y está definido por **<cstring>**.

El encabezado **<cstring>** también define varias funciones que empiezan con el prefijo "mem". Estas funciones operan sobre caracteres, pero no usan la convención de terminación en carácter nulo. En ocasiones son útiles cuando se manipulan cadenas y también pueden utilizarse para otros fines. Las funciones son **memchr()**, **memcmp()**, **memcpy()**, **memmove()** y **memset()**. Las primeras tres operan de manera similar a **strchr()**, **strcmp()**, **strcpy()**, respectivamente, excepto porque toman un parámetro adicional que especifica el número de caracteres en que operan. La función **memset()** asigna un valor específico a un bloque de memoria. La función **memmove()** mueve un bloque de

caracteres. A diferencia de **memcpy()**, **memmove()** puede utilizarse para mover caracteres en matrices que se superponen. Es la única función "mem" empleada en este capítulo y se muestra aquí:

void *memmove(void **destino*, const void **origen*, size_t, *cuenta*)

Copia *cuenta* caracteres de la matriz señalada por *origen* en la señalada por *destino*. Devuelve *destino*. Como se mencionó, la copia se realiza correctamente, aunque se superpongan las matrices. Sin embargo, en este caso, la matriz señalada por *origen* puede modificarse (aunque *origen* esté especificado como **const**).

NOTA *Visual C++ de Microsoft "descontinúa" (ya no recomienda el uso de) varias funciones de cadena estándar, como **strcpy()**, por razones de seguridad. Por ejemplo, Microsoft recomienda, en cambio, el uso de **strcpy_s()**. Sin embargo, estas opciones no están definidas por el estándar de C++ y no son estándares. Por tanto, en este libro se utilizarán las funciones especificadas por el estándar internacional para C++.*

Revisión general de la clase string

Aunque las cadenas terminadas en un carácter nulo son muy eficientes, experimentan dos problemas. En primer lugar, no definen un tipo. Es decir, la representación de una cadena como una matriz de caracteres terminados por un carácter nulo es una *convención*. Aunque ésta es bien comprendida y tiene un amplio reconocimiento, no es un tipo de datos, en el sentido normal. (En otras palabras, la cadena terminada en un carácter nulo no es parte del sistema de tipo de C++.) Como resultado, este tipo de carpetas no puede manipularse con operadores. Por ejemplo, *no puede* unir dos cadenas terminadas en un carácter nulo al usar el operador + o = para asignar una cadena terminada en un carácter nulo a otra. Por tanto, la siguiente secuencia no funcionará:

```
// Esta secuencia es un error.
char cadA[] = "alfa";
char cadB[] = "beta";
char cadC[9] = cadA + cadB; // ¡Perdón! ¡No funciona!
```

En cambio, debe usar llamadas a funciones de biblioteca para realizar estas operaciones, como se muestra a continuación:

```
// Esta secuencia sí funciona.
char cadA[] = "alfa";
char cadB[] = "beta";
char cadC[9];
strcpy(cadC, cadA);
strcat(cadB, cadA);
```

Esta secuencia correcta usa **strcpy()** y **strcat()** para asignar a **cadC** una cadena que contiene la unión de **cadA** y **cadB**. Aunque logra el resultado deseado, la manipulación de cadenas mediante el uso de funciones en lugar de operadores hace que aun las operaciones más rudimentarias sean un poco confusas.

El segundo problema con las cadenas terminadas en un carácter nulo es la facilidad con que pueden crearse errores. En las manos de un programador inexperto o descuidado, es muy fácil sobrepasar el final de la matriz que contiene una cadena. Debido a que C++ no proporciona comprobación de límites en las operaciones con matrices (o apuntadores), no hay nada que evite que se rebase el final de una matriz. Por tanto, si la cadena de origen contiene más caracteres de los que puede contener la matriz de destino, ésta se desbordará. En el mejor de los casos, un desbordamiento de una matriz simplemente hará que deje de funcionar el programa. Sin embargo, en el peor de los casos, da como resultado una brecha de seguridad basada en el ahora notorio ataque "desbordamiento de búfer".

Debido al deseo de integrar cadenas en el sistema general de tipos de C++ y para evitar el desbordamiento de matrices, se añadió a C++ un tipo de datos de cadena. Está basado en la clase de plantilla **basic_string**, que está declarado en el encabezado **<string>**. Como se mencionó, hay dos especializaciones de esta clase: **string** y **wstring**, que también se declaran en **<string>**. La clase **string** es para cadenas **char**. La clase **wstring** es para cadena de caracteres ampliados basada en **wchar_t**. Aparte del tipo de caracteres, las dos especializaciones funcionan, en esencia, de la misma manera. Debido a que las cadenas **char** son, por mucho, las que se usan con más frecuencia, el siguiente análisis y soluciones utilizan **string**, pero casi toda la información puede adaptarse fácilmente a **wstring**.

La clase **string** crea un tipo de datos dinámico. Esto significa que una instancia de **string** puede crecer lo necesario durante el tiempo de ejecución para adaptarse a un aumento en la longitud de la cadena. Esto no sólo elimina el problema de desbordamiento del búfer, sino que lo libera de tener que preocuparse por especificar la longitud correcta de una cadena. La clase **string** maneja esto automáticamente.

La clase **string** define varios constructores y muchas funciones. He aquí tres constructores de uso común:

string(const Allocator %*asig* = Allocator())
string(const char **cad*, const Allocator %*asig* = Allocator())
string(const string &*cad*, size_type *ind_inicio* = 0, size_tipe *num* = npos,
        const Allocator %*asig* = Allocator())

La primera forma crea un objeto **string** vacío. La segunda crea un objeto **string** a partir de la cadena terminada en un carácter nulo señalada por *cad*. Esta forma le permite crear una **string** a partir de una cadena terminada en un carácter nulo. La tercera forma crea una **string** a partir de otra **string**. La cadena creada contiene *num* caracteres de *cad*, empezando en el índice especificado por *ind_inicio*. Con frecuencia, en el tercer constructor se permiten los parámetros *ind_inicio* y *num*, como opción predeterminada. En este caso, *ind_inicio* contiene cero (lo que indica el inicio de la cadena) y *num* contiene el valor **npos**, que indica (en este caso) la longitud de la cadena más larga posible. En todos los casos, observe que los constructores permiten que se especifique el asignador. Se trata de un objeto de tipo **Allocator** que proporciona asignación de memoria a la cadena. Con mayor frecuencia, este argumento se permite como opción predeterminada, lo que da como resultado el uso del asignador predeterminado.

He aquí el aspecto del constructor cuando se usan los valores predeterminados del argumento, lo que sucede con frecuencia:

string()
string(const char **cad*)
string(const string &*cad*)

Todos ellos usan el asignador predeterminado. El primero crea una cadena vacía. El segundo y tercero crean una que contiene *cad*.

La clase **string** define muchas funciones, y casi todas tienen varias formas de sobrecarga. Por tanto, no resulta práctica una descripción completa de cada función de **string**. En cambio, las soluciones individuales describen con detalle las funciones que emplean. Sin embargo, para darle una idea del poder que tiene a su disposición con **string**, he aquí una lista de funciones esenciales, agrupadas en categorías.

En las siguientes funciones se busca el contenido de una cadena:

find	Devuelve el índice en que se encuentra la primera aparición de una subcadena o un carácter dentro de la cadena que invoca. Devuelve **npos** si no se encuentra una coincidencia.
rfind	Devuelve el índice en que se encuentra la última aparición de una subcadena o un carácter dentro de la cadena que invoca. Devuelve **npos** si no se encuentra una coincidencia.
find_first_of	Busca en la cadena que invoca la primera aparición de cualquier carácter contenido dentro de una segunda cadena y devuelve el índice en que se encuentra la coincidencia dentro de la cadena que invoca. Devuelve **npos** si no se encuentra una coincidencia.
find_last_of	Busca en la cadena que invoca la última aparición de cualquier carácter contenido dentro de una segunda cadena y devuelve el índice en que se encuentra la coincidencia dentro de la cadena que invoca. Devuelve **npos** si no se encuentra una coincidencia.
find_first_not_of	Busca en la cadena que invoca la primera aparición de cualquier carácter que *no* está contenido dentro de una segunda cadena y devuelve el índice en que se encuentra la coincidencia dentro de la cadena que invoca. Devuelve **npos** si no se encuentra una coincidencia.
find_last_not_of	Busca en la cadena que invoca la última aparición de cualquier carácter que *no* está contenido dentro de una segunda cadena y devuelve el índice en que se encuentra la coincidencia dentro de la cadena que invoca. Devuelve **npos** si no se encuentra una coincidencia.

El siguiente conjunto de funciones de cadena modifica el contenido de una cadena:

append	Añade una cadena al final de la cadena que invoca.
assign	Asigna una nueva cadena a la cadena que invoca.
clear	Elimina todos los caracteres de la cadena que invoca.
copy	Copia un rango de caracteres de la cadena que invoca en una matriz.
erase	Elimina uno o más caracteres de la cadena que invoca.
insert	Inserta una cadena, subcadena o uno o más caracteres en la cadena que invoca.
push_back	Agrega un carácter al final de la cadena que invoca.
replace	Reemplaza una parte de la cadena que invoca.
resize	Disminuye o alarga la cadena que invoca. Cuando se acorta, es posible que se pierdan caracteres.
swap	Intercambia dos cadenas.

Las siguientes funciones devuelven información acerca de un objeto **string**:

capacity	Devuelve el número de caracteres que la cadena que invoca puede contener sin que se asigne más memoria.
c_str	Devuelve un apuntador a una cadena terminada en un carácter nulo que contiene los mismos caracteres que los contenidos en la cadena que invoca.
data	Devuelve un apuntador a una matriz que contiene los caracteres en la cadena que invoca. Esta matriz no termina en un carácter nulo.
empty	Devuelve true si la cadena que invoca está vacía.
length	Devuelve el número de caracteres contenido en la cadena que invoca.
max_size	Devuelve el tamaño máximo de una cadena.
size	Igual que length.

El siguiente conjunto de funciones da soporte a iteradores:

begin	Devuelve un iterador al principio de la cadena.
end	Devuelve un iterador a la ubicación en que pasa el final de la cadena.
rbegin	Devuelve un iterador inverso al final de una cadena.
rend	Devuelve un iterador inverso al lugar que se encuentra uno antes del inicio de la cadena.

Las siguientes dos funciones obtienen una subcadena o un carácter de una cadena:

at	Devuelve una referencia al carácter en un índice especificado dentro de la cadena que invoca.
substr	Devuelve una cadena que es una subcadena de la que invoca. Se especifican el índice inicial y el número de caracteres en la subcadena.

Además de las funciones que se acaban de mostrar, hay dos más. Puede comparar dos cadenas al llamar a **compare()**. Puede hacer que una cadena asigne memoria suficiente para contener un número de caracteres específico al llamar a **reverse()**. Debido a que **string** es una estructura de datos dinámica, la asignación previa de memoria evita la necesidad de costosas reasignaciones a medida que aumenta el tamaño de la cadena. Por supuesto, esto sólo resulta útil si sabe de antemano el tamaño de la cadena más larga.

La clase **string** también define varios tipos, incluido **size_type**, que es una forma de entero no asignado que puede contener un valor igual a la longitud de la cadena más larga a la que da soporte la implementación. El tipo de carácter contenido por la cadena está definido por **value_type**. La clase **string** también declara varios tipos de iterador, incluido **iterator** y **reverse_iterator**.

La clase **string** declara una variable **static const**, llamada **npos**, de tipo **size_type**. Este valor está inicializado en –1. Esto da como resultado un **npos** que contiene el valor no asignado más grande que **size_type** puede representar. Por tanto, en todos los casos, **npos** representa un valor que es por lo menos uno más largo que el tamaño de la cadena más larga. La variable **npos** suele usarse para indicar la condición "final de la cadena". Por ejemplo, si una búsqueda falla, se devuelve **npos**. También se utiliza para solicitar que alguna operación tenga lugar hasta el final de una cadena.

Se han sobrecargado varias operaciones para aplicar a objetos de cadena. Se muestran a continuación:

Operador	Significado
=	Asignación
+	Unión
+=	Asignación de unión
==	Igualdad
!=	Desigualdad
<	Menor que
<=	Menor que o igual a
>	Mayor que
>=	Mayor que o igual a
[]	Subíndices
<<	Salida
>>	Entrada

Estos operadores le permiten usar objetos **string** en expresiones y eliminar la necesidad de llamadas a funciones como **strcpy()**, **strcat()** o **strcmp()**, que se requieren para cadenas terminadas en un carácter nulo. Por ejemplo, puede usar un operador de relación como **<** para comparar dos objetos **string**, asignar un objeto **string** a otro al usar el operador **=** y unir dos objetos de cadena con el operador **+**.

En general, puede combinar objetos **string** con cadenas terminadas en un carácter nulo dentro de una expresión, siempre y cuando el resultado deseado sea un objeto **string**. Por ejemplo, el operador **+** puede usarse para unir un objeto **string** con otro o un objeto **string** con una cadena estilo C. Es decir, tienen soporte las siguientes variaciones:

cadena + cadena
cadena + cadena C
cadena C + cadena

Además, puede usar **=** para asignar una cadena terminada en un carácter nulo a un objeto **string** o comparar ambos mediante operadores relacionales.

Hay otro aspecto importante en la clase **string**: también es un contenedor compatible con STL. La clase **string** da soporte a iteradores y funciones como **begin()**, **end()** y **size()**, que deben implementar todos los contenedores. Debido a que **string** es un contenedor, es compatible con los otros contenedores estándar, como **vector**. También puede operarse mediante los algoritmos STL. Esto le da una capacidad y flexibilidad extraordinarias cuando se manejan cadenas.

Tomada como un todo, la clase **string** hace que el manejo de cadena sea excesivamente conveniente y libre de problemas. Puede realizar operaciones con cadenas más comunes mediante operadores, y una serie rica en funciones miembro de **string** como búsqueda, reemplazo y comparación de cadena fácil y relativamente libre de errores. No es necesario que se preocupe por desbordar una matriz, por ejemplo, cuando asigna una cadena a otra. En general, el tipo **string** ofrece seguridad y conveniencia que excede en mucho la de cadenas terminadas en un carácter nulo.

A pesar de las ventajas de la clase **string**, las cadenas terminadas en un carácter nulo se usan ampliamente en C++. Una razón de esto es que (como se explicó antes) las literales de cadena son cadenas terminadas en carácter nulo. Otra razón es que todo el poder de **string** tiene un precio. En algunos casos, las operaciones con objetos **string** son más lentas que las operaciones terminadas en un carácter nulo. Por tanto, para aplicaciones en que el alto desempeño es una aplicación importante y no se requieren los beneficios de una **string**, las cadenas terminadas en un carácter nulo son todavía una buena elección. Es importante aclarar, sin embargo, que para muchas otras aplicaciones, la clase **string** es todavía la mejor elección.

Excepciones de cadenas

Aunque el manejo de cadenas mediante **string** evita muchos de los accidentes comunes con cadenas terminadas en un carácter nulo, aún es posible que se generen errores. Por fortuna, cuando ocurre un error al manipular un objeto **string**, se obtiene una excepción, en lugar de que un programa deje de funcionar o se produzca una brecha de seguridad. Esto le da una oportunidad de rectificar el error, o por lo menos de realizar un apagado ordenado.

Hay dos excepciones que pueden generarse cuando se trabaja con objetos **string**. La primera es **lenght_error**. Ésta se lanza cuando se hace un intento de crear una cadena más larga que la cadena más larga posible. Esto podría suceder en varios casos diferentes, como cuando se unen cadenas o se inserta una subcadena en otra. La longitud de la cadena más larga posible se encuentra al llamar a la función **max_size()**. La segunda excepción es **out_of_range**. Se lanza cuando un argumento está fuera de rango. Ambas excepciones se declaran en **<stdexcept>**. Debido a que ninguno de los ejemplos de este capítulo genera estas excepciones, los ejemplos no los manejan de manera explícita. Sin embargo, en sus propias aplicaciones, tal vez necesite hacerlo.

Realice operaciones básicas en cadenas terminadas en un carácter nulo

Componentes clave		
Encabezado	**Clases**	**Funciones**
<cstring>		char *strcat(char **cad1*, const char **cad2*)
		int strcmp(const char **cad1*, const char **cad2*)
		char *strcpy(char **destino*, const char **origen*)
		size_t strlen(const char **cad*)

En esta solución se muestra cómo realizar las siguientes operaciones básicas con cadenas terminadas en un carácter nulo:

- Obtener la longitud de una cadena
- Copiar una cadena
- Unir una cadena al final de otra
- Comparar dos cadenas

Hay dos operaciones que suelen ser necesarias cada vez que se usan cadenas terminadas en un carácter nulo en un programa de C++. Serán familiares para muchos lectores (sobre todo quienes tienen antecedentes en programación en C). Se empieza con ellas porque ilustran conceptos fundamentales relacionados con el trabajo con este tipo de cadenas. También ilustran por qué debe tener cuidado con evitar desbordamientos de búfer cuando use cadenas terminadas en un carácter nulo.

Paso a paso

Para realizar las operaciones básicas con cadenas terminadas en un carácter nulo se requieren estos pasos:

1. Incluir el encabezado **<cstring>**.
2. Para obtener la longitud de la cadena, llame a **strlen()**.
3. Para copiar una cadena en otra, llame a **strcpy()**.
4. Para unir una cadena al final de otra, llame a **strcat()**.
5. Para comparar dos cadenas, llame a **strcmp()**.

Análisis

Las funciones que dan soporte a cadenas terminadas en un carácter nulo se declaran en el encabezado **<cstring>**. Por tanto, un programa que utiliza éstas u otras funciones que operan en cadenas terminadas en un carácter nulo, debe incluir este encabezado.

Para obtener la longitud de una cadena terminada en un carácter nulo, llame a **strlen()**, que se muestra aquí:

size_t strlen(const char **cad*)

Devuelve el número de caracteres en la cadena señalada por *cad*. Como se explicó en la revisión general, una cadena terminada en un carácter nulo es simplemente una matriz de caracteres que cumple esta condición. El valor devuelto por **strlen()** no incluye el terminador de carácter nulo. Por tanto, la cadena **"prueba"** tiene una longitud de 6. Sin embargo, comprenda que la matriz que contendrá **"prueba"** debe tener por lo menos 7 caracteres de largo para que haya espacio para el terminador de carácter nulo. El tipo **size_t** es alguna forma de entero no asignado que puede representar el resultado de las operaciones de **sizeof**. Por tanto, es un tipo que puede representar la longitud de la cadena más larga.

Para copiar una cadena terminada en un carácter nulo en otra, se usa **strcpy()**, que se muestra a continuación:

char *strcpy(char **destino*, const char **origen*)

Esta función copia los caracteres en la cadena señalada por *origen* en la matriz señalada por *destino*. El resultado termina en un carácter nulo. En todos los casos, debe asegurarse de que la matriz señalada por *destino* es lo suficientemente larga para contener los caracteres señalados por *origen*. Si no, la copia sobrescribirá el final de la matriz de destino. Esto corromperá su programa y es una manera en que puede generarse el famoso "ataque de desbordamiento de búfer". Esta función devuelve *destino*.

Para unir una cadena terminada en un carácter nulo con el final de otra, se llama a **strcat()**:

char *strcat(char **cad1*, const char **cad2*)

Esta función copia los caracteres en la cadena a la que señala *cad2* al final de la cadena a la que señala *cad1*. La cadena resultante termina en un carácter nulo. Es fundamental que la matriz a la que señala *cad1* sea lo suficientemente larga para contener la cadena resultante. De lo contrario, se presentará un desbordamiento de matriz. Esto corromperá su programa y es otra manera de que pueda presentarse un ataque de desbordamiento de búfer. La función devuelve *cad1*.

Puede comparar lexicográficamente (mediante el orden del diccionario) dos cadenas empleando **strcmp()**, que se muestra a continuación:

int strcmp(const char **cad1*, const char **cad2*)

Devuelve cero si las dos cadenas son iguales. De otra manera, devuelve menos de cero si la cadena a la que apunta *cad1* es menor que la señalada por *cad2* y mayor que cero si la cadena a la que apunta *cad1* es mayor que la señalada por *cad2*. La comparación es sensible a mayúsculas y minúsculas.

Ejemplo

En el siguiente ejemplo se muestran **strcpy()**, **strcat()** y **strlen()** en acción:

```
// Demuestra las funciones básicas de cadena terminada en un carácter nulo.
#include <iostream>
#include <cstring>

using namespace std;

int main() {
    char cadA[10] = "Gato";
    char cadB[6] = "Cebra";
    char cadC[6] = "Cabra";
    char cadD[7] = "Jirafa";

    cout << "Las cadenas son: " << endl;
    cout << "cadA: " << cadA << endl;
    cout << "cadB: " << cadB << endl;
    cout << "cadC: " << cadC << endl;
    cout << "cadD: " << cadD << "\n\n";

    // Despliega la longitud de cadA.
    cout << "La longitud de cadA es " << strlen(cadA) << endl;

    // Une cadB con cadA.
    strcat(cadA, cadB);
    cout << "cadA una vez unida: " << cadA << endl;
    cout << "La longitud de cadA es ahora " << strlen(cadA) << endl;

    // Copia cadC en cadB.
    strcpy(cadB, cadC);
    cout << "cadB contiene ahora: " << cadB << endl;

    // Compara cadenas.
    if(!strcmp(cadB, cadC))
```

```
    cout << "cadB es igual a cadC\n";
  int resultado = strcmp(cadC, cadD);
  if(!resultado)
    cout << "cadC es igual a cadD\n";
  else if(resultado < 0)
    cout << "cadC es menor que cadD\n";
  else if(resultado > 0)
  cout << "cadC es mayor que cadD\n";

  return 0;
}
```

Aquí se muestra la salida:

```
Las cadenas son:
cadA: Gato
cadB: Cebra
cadC: Cabra
cadD: Jirafa

La longitud de cadA es 4
cadA una vez unida: GatoCebra
La longitud de cadA es ahora 9
cadB contiene ahora: Cabra
cadB es igual a cadC
cadC es menor que cadD
```

Observe cómo se declaró que la matriz que contiene **cadA** es mayor de lo necesario para contener su cadena inicial. Este espacio adicional le permite acomodar la unión de **cadB**. Además, observe cómo **cadB** y **cadC** tienen el mismo tamaño. Esto permite copiar el contenido de **cadC** en **cadB**. Recuerde, en todos los casos, que la matriz que recibe el resultado de una copia o unión de una cadena debe ser lo suficientemente larga. Por ejemplo, en el programa anterior, tratar de copiar **cadD** en **cadC** causaría un error, porque **cadC** sólo tiene seis elementos de largo, pero **cadD** requiere siete (seis para los caracteres de Jirafa y uno para el terminado de carácter nulo).

Opciones

En casos en que no sabe en tiempo de compilación si la longitud de la matriz de destino es suficiente para contener el resultado de una copia o unión de cadena, necesitará confirmar ese hecho en tiempo de ejecución antes de tratar la operación. Una manera de hacer esto es usar **sizeof** para determinar el tamaño de la matriz de destino. Por ejemplo, suponiendo el programa de ejemplo anterior, he aquí la manera de agregar una "revisión de seguridad" que asegura que **cadA** es lo bastante larga para contener la unión de **cadA** y **cadB**:

```
if(sizeof(cadA) > strlen(cadA) + strlen(cadB)) strcat(cadA, cadB);
```

Aquí, el tamaño de la matriz de destino se obtiene al llamar a **sizeof** en la matriz. Esto devuelve la longitud de la matriz en bytes, lo que en matrices de tipo **char** es igual al número de caracteres en la matriz. Este valor debe ser mayor que la suma de las dos cadenas que se unirán. (Recuerde que se necesita un carácter adicional para contener el terminador de carácter nulo.) Al usar este método, asegura que la matriz de destino no se desbordará.

NOTA *La técnica anterior para evitar un desbordamiento de matriz funciona para cadenas* ***char****, no para cadenas* ***wchar_t****. En el caso de estas últimas, no necesita usar una expresión como*

```
if(sizeof(cadA) > wcslen(cadA) *sizeof(wchar_t) +
      wcslen(cadB) *sizeof(wchar_t)) // . . .
```

Esto toma en consideración el tamaño de un carácter ampliado.

En ocasiones tal vez quiera operar sólo en una parte de una cadena, en lugar de toda ella. Por ejemplo, tal vez quiera copiar sólo una parte de la cadena a otra o comparar sólo una parte de dos cadenas. C++ incluye funciones que manejan estos tipos de situaciones. Son **strncpy()**, **strncat()** y **strncmp()**. Cada una se describe a continuación.

Para copiar sólo una porción de una cadena en otra, use **strncpy**, mostrado aquí:

char *strncpy (char **destino*, const char **origen*, size_t *cuenta*)

La función no copia más de *cuenta* caracteres de *origen* a *destino*. Si *origen* contiene menos de *cuenta* caracteres, los caracteres nulos se adjuntarán al final de *destino* hasta que se hayan copiado *cuenta* caracteres. Sin embargo, si la cadena señalada por *origen* es más larga que *cuenta* caracteres, la cadena resultante señalada por *destino* no terminará en un carácter nulo. Devuelve *destino*.

Puede unir sólo una parte de una cadena a otra al llamar a **strncat()**, que se muestra a continuación:

char *strncat(char **cad1*, const char **cad2*, size_t *cuenta*)

Une no más de *cuenta* caracteres de la cadena señalada por *cad2* al final de *cad1*. Devuelve *cad1*.

Para comparar una parte de una cadena a otra, use **strncmp()**, que se muestra a continuación:

int strncmp(const char **cad1*, const char **cad2*, size_t *cuenta*)

La función **strncmp()** compara no más de los primeros *cuenta* caracteres en la cadena a la que señala *cad1* con los de la cadena a la que señala *cad2*. Devuelve menos de cero si *cad1* es menor que *cad2*, mayor que cero si *cad1* es mayor que *cad2*, y cero si las dos cadenas son iguales.

Busque una cadena terminada en un carácter nulo

Componentes clave		
Encabezado	**Clases**	**Funciones**
<cstring>		char *strchr(const char **cad*, int *car*)
		char *strpbrk(const char **cad1*, const char **cad2*)
		char *strstr(const char **cad1*, const char **cad2*)

Otra parte común del manejo de cadenas incluye la búsqueda. He aquí tres ejemplos. Tal vez quiera saber si una cadena contiene la subcadena ".com" o ".net" cuando se procesa una dirección

de Internet. Quizá desee encontrar el primer punto en un nombre de archivo, de modo que puede dar soporte al nombre de archivo a partir de su extensión. Tal vez quiera explorar un registro de facturas para encontrar la cadena "Vencido" para que pueda contar el número de cuentas vencidas. Para manejar estos tipos de tareas, C++ proporciona funciones que buscan una cadena terminada en un carácter nulo. En esta solución se muestran varias de ellas. De manera específica, muestra cómo buscar un carácter determinado, cualquier conjunto de caracteres o una subcadena en una cadena.

Paso a paso

Para buscar una cadena se requieren los siguientes pasos:

1. Para buscar un carácter específico, llame a **strchr()**.
2. Para buscar cualquier carácter en un conjunto de éstos, llame a **strpbrk()**.
3. Para buscar una subcadena, llame a **strstr()**.

Análisis

Para encontrar la primera aparición de un carácter determinado dentro de una cadena, llame a **strchr()** que se muestra aquí:

char *strchr(const char **cad*, int *car*)

Devuelve un apuntador a la primera aparición del byte de orden bajo de *car* en la cadena señalada por *cad*. Si no se encuentra una coincidencia, se devuelve un apuntador nulo.

Para encontrar la primera aparición de cualquier carácter dentro de un conjunto de caracteres, llame a **strpbrk()**, que se muestra a continuación:

char *strpbrk(const char **cad1*, const char **cad2*)

Esta función devuelve un apuntador al primer carácter en la cadena a la que señala *cad1* y que coincide con *cualquier carácter* en la cadena señalada por *cad2*. Si no se encuentran coincidencias, se devuelve un apuntador nulo.

Para encontrar la primera aparición de una subcadena determinada dentro de una cadena, llame a **strstr()** que se muestra aquí:

char *strstr(const char **cad1*, const char **cad2*)

Devuelve un apuntador a la primera aparición de la cadena que señala *cad2* dentro de la cadena señalada por *cad1*. Si no se encuentra una coincidencia, se devuelve un apuntador nulo.

Ejemplo

En el siguiente ejemplo se demuestran **strchr()**, **strpbrk()** y **strstr()**:

```
// Busca una cadena terminada en un carácter nulo.
#include <iostream>
#include <cstring>

using namespace std;

int main() {
```

```
  const char *url = "HerbSchildt.com";
  const char *url2 = "Apache.org";
  const char *diremail = "Herb@HerbSchildt.com";

  const char *tld[] = { ".com", ".net", ".org" };

  const char *p;

  // Primero, determina si url y url2 contienen .com, .net u .org.
  for(int i=0; i < 3; i++) {
    p = strstr(url, tld[i]);
    if(p) cout << url << " tiene el dominio de nivel superior " << tld[i] << endl;

    p = strstr(url2, tld[i]);
    if(p) cout << url2 << " tiene el dominio de nivel superior " << tld[i] << endl;
  }

  // Busca un carácter específico.
  p = strchr(diremail, '@');
  if(p) cout << "El nombre del sitio de la direcci\u00a2n de correo electr\
u00a2nico es: " << p+1 << endl;

  // Busca un carácter entre un conjunto de ellos.
  // En este caso, encuentra el primer @ o punto.
  p = strpbrk(diremail, "@.");

  if(p) cout << "Se encontr\u00a2 " << *p << endl;

  return 0;
}
```

En el código anterior, observará el uso de la secuencia de escape "\u00a2" para la "o". Es indispensable el uso de estas secuencias para el despliegue de caracteres especiales como á o ñ en la salida del programa. Aquí se muestra la salida:

```
HerbSchildt.com tiene el dominio de nivel superior .com
Apache.org tiene el dominio de nivel superior .org
El nombre del sitio de la dirección de correo electrónico es: HerbSchildt.com
Se encontró @
```

Opciones

Además de la búsqueda de funciones usada en esta solución, hay varias otras a las que da soporte C++. Dos que resultan especialmente útiles en algunos casos son **strspn()** y **strcspn()**. Aquí se muestran:

size_t strspn(const char **cad1*, const char **cad2*)
size_t strcspn(const char **cad1*, const char **cad2*)

La función **strspn()** devuelve el índice del primer carácter en la cadena señalada por *cad1* que *no* coincide con cualquiera de los caracteres en la cadena a la que apunta *cad2*. La función **strcspn()** devuelve el índice del primer carácter en la cadena señalada por *cad1* que coincide con cualquier carácter en la cadena señalada por *cad2*.

Puede encontrar la última aparición de un carácter dentro de una cadena terminada en un carácter nulo al llamar a **strrchr()**:

char *strrchar(const char **cad*, int *car*)

Devuelve un apuntador a la última aparición del byte de orden bajo de *car* en la cadena señalada por *cad*. Si no se encuentra una coincidencia, se devuelve un apuntador nulo.

La función **strtok()** también se utiliza para buscar una cadena. Se describe en su propia solución. Consulte *Convierta en fichas una cadena terminada en un carácter nulo*.

Invierta una cadena terminada en un carácter nulo

Componentes clave		
Encabezado	**Clases**	**Funciones**
<cstring>		size_t strlen(char **cad*)

En esta solución se muestra cómo realizar una tarea simple, pero útil: revertir una cadena terminada en un carácter nulo. Aunque la inversión de una cadena es una operación fácil para el programador experimentado, es una fuente común de preguntas para el principiante. Por esta sola razón merece su inclusión en este libro. Sin embargo, hay otras varias razones para incluirla. En primer lugar, hay muchas maneras de invertir una cadena, y cada variación ilustra una técnica diferente para manejar una cadena terminada en un carácter nulo. En segundo lugar, el mecanismo básico usado para invertir una cadena puede adaptarse a otros tipos de manipulaciones de cadena. Por último, demuestra en términos muy prácticos cómo manejar cadenas terminadas en un carácter nulo suele depender de código práctico de muy bajo nivel. A menudo, este código puede ser muy eficiente, pero requiere más trabajo que el uso de la clase **string**.

En la solución mostrada aquí se invierte la cadena. Esto significa que se modifica la cadena original. Por lo general, esto es lo que se necesita. Sin embargo, en la sección *Opciones* se muestra una variación que crea una copia inversa de la cadena.

Paso a paso

Hay muchas maneras de afrontar la tarea de invertir una cadena. En esta solución se usa un método simple pero efectivo que está basado en el intercambio de extremo a extremo de los caracteres correspondientes de la cadena. Se pone este código dentro de una función llamada **invcad()**.

1. Cree una función llamada **invcad()** que tenga este prototipo:

   ```
   void invcad(char *cad)
   ```

   La cadena que habrá de invertirse se pasa a **cad**.
2. Dentro de **invcad()**, cree un bucle **for** que controle las dos variables que se usarán para indizar la matriz que contiene la cadena. Inicialice la primera variable en cero y auméntela

cada vez que se recorra el bucle. Inicialice la segunda variable en el índice del último carácter de la cadena y disminúyalo con cada iteración. Este valor se obtiene al llamar a **strlen()**.
3. Con cada paso que se recorra el bucle, intercambie los caracteres en los dos índices.
4. Detenga el bucle cuando el primer índice sea igual o mayor que el segundo índice. En este punto, se invertirá la cadena.

Análisis

Como la mayoría de los lectores sabe, cuando se usa un nombre de matriz por sí solo, sin un índice, representa un apuntador a la matriz. Por tanto, cuando pasa una matriz a una función, en realidad sólo está pasando un apuntador a esa matriz. Esto significa que una función que recibirá una cadena terminada en un carácter nulo como argumento debe declarar que su parámetro es de tipo **char** *. Por eso, el parámetro **cad** de **invcad()** se declara como **char *cad**.

Aunque **cad** es un apuntador, puede indizarse como una matriz, empleando la sintaxis normal de indización de matriz. Para invertir el contenido de una cadena, cree un bucle **for** que controla dos variables, que sirven como índices en la cadena. Un índice empieza en cero e indiza a partir del principio de la cadena. El otro índice empieza en el último carácter de la cadena. Cada vez que recorra el bucle, se intercambian los caracteres que se encuentran en los índices especificados. Luego, el primer índice se aumenta y se reduce el segundo. Cuando los índices convergen (es decir, cuando el primer índice es igual o mayor que el segundo), la cadena se invierte. He aquí la manera de escribir este bucle:

```
int i, j;
char t;

for(i = 0, j = strlen(cad)-1; i < j; ++i, --j) {
  t = cad[i];
  cad[i] = cad[j];
  cad[j] = t;
}
```

Observe que el índice del último carácter de la cadena se obtiene al restar uno del valor devuelto por **strlen()**. Aquí se muestra su prototipo:

size_t strlen(const char **cad*)

La función **strlen()** devuelve la longitud de una cadena terminada en un carácter nulo, que es el número de caracteres en la cadena. Sin embargo, no se cuenta el terminador de carácter nulo. Debido a que el indizado de la matriz en C++ empieza en cero, debe restarse 1 a este valor para obtener el índice del último carácter en la cadena.

Ejemplo

Uniendo las piezas, he aquí una manera de escribir la función **invcad()**:

```
// Invierte una cadena en el lugar.
void invcad(char *cad) {
  int i, j;
  char t;

  for(i = 0, j = strlen(cad)-1; i < j; ++i, --j) {
```

```
    t = cad[i];
    cad[i] = cad[j];
    cad[j] = t;
  }
}
```

En el siguiente programa se muestra **invcad()** en acción:

```
// Invierte una cadena en el lugar.
#include <iostream>
#include <cstring>

using namespace std;

void invcad(char *cad);

int main() {
  char cad[] = "abcdefghijklmnopqrstuvwxyz";

  cout << "Cadena original: " << cad << endl;

  invcad(cad);

  cout << "Cadena invertida: " << cad << endl;

  return 0;
}
 // Invierte una cadena en el lugar.
void invcad(char *cad) {
  int i, j;
  char t;

  for(i = 0, j = strlen(cad)-1; i < j; ++i, --j) {
    t = cad[i];
    cad[i] = cad[j];
    cad[j] = t;
  }
}
```

Aquí se muestra la salida:

```
Cadena original: abcdefghijklmnopqrstuvwxyz
Cadena invertida: zyxwvutsrqponmlkjihgfedcba
```

Opciones

Aunque invertir una cadena terminada en un carácter nulo es una tarea simple, permite algunas variaciones interesantes. Por ejemplo, el método usado en la solución depende de la indización de la matriz, que tal vez es la manera más clara de implementar esta función. Sin embargo, quizás no sea la más eficiente. Una opción consiste en usar apuntadores en lugar de indización de matriz. Dependiendo del compilador que está usando (y las optimizaciones activadas), las operaciones de apuntador pueden ser más rápidas que la indización de matriz. Además, muchos programadores simplemente prefieren el uso de apuntadores en lugar de indización de matriz cuando se recorre en ciclo una matriz de manera estrictamente secuencial. Cualquiera que sea la razón, la versión

de apuntador es fácil de implementar. He aquí una manera de retrabajar **invcad()**, de modo que sustituye las operaciones de apuntador para indización de matriz:

```
// Invierte una cadena en el lugar. Use apuntadores en lugar de indización de
matriz.
void invcad(char *cad) {
  char t;

  char *inc_p = cad;
  char *dec_p = &cad[strlen(cad)-1];

  while(inc_p <= dec_p) {
    t = *inc_p;
    *inc_p++ = *dec_p;
    *dec_p-- = t;
  }
}
```

Una de las maneras más interesantes de revertir una cadena emplea la recursión. He aquí una implementación:

```
// Invierte una cadena en el lugar al usar recursión.
void invcad_r(char *cad) {
  invcad_recursiva(cad, 0, strlen(cad)-1);
}

// A esta función se le llama con un apuntador a la cadena que se
// invertirá y los índices de principio y final de los caracteres
// que se invertirán. Por tanto, su primera llamada pasa cero
// para inicio y strlen(cad)-1 para final. La posición del
// terminador del carácter nulo no cambia.
void invcad_recursiva(char *cad, int inicio, int final) {
  if(inicio < final)
    invcad_recursiva(cad, inicio+1, final-1);
  else
    return;

  char t = cad[inicio];
  cad[inicio] = cad[final];
  cad[final] = t;
}
```

Observe que **invcad_r()** llama a **invcad_recursiva()** para revertir la cadena. Esto permite que se llame a **invcad_r()** únicamente con un apuntador a la cadena. Observe cómo las llamadas recursivas invierten la cadena. Cuando **inicio** es menor que **final**, se hace una llamada recursiva a **invcad_recursiva()**; y el índice inicial aumenta en uno y el índice final disminuye en uno. Cuando estos dos índices se unen, se ejecuta la instrucción **return**. Esto causa que las llamadas recursivas empiecen a devolverse, mientras se intercambian los correspondientes caracteres. Como algo interesante, puede usarse la misma técnica general para invertir el contenido de cualquier tipo de matriz. Su uso en una cadena terminada en un carácter nulo es simplemente un caso especial.

La última opción presentada aquí funciona de manera diferente de los métodos anteriores, porque crea una copia de la cadena original que contiene el inverso de la cadena original. Por tanto, deja ésta sin cambio. Esta técnica es útil cuando no debe modificarse la cadena original.

```
// Hace una copia inversa de una cadena.
void invcadcopia(char *cadr, const char *cadorg) {

  cadr += strlen(cadorg);
  *cadr-- = '\0';

  while(*cadorg)  *cadr-- = *cadorg++;
}
```

A esta función se le pasa un apuntador a la cad original en **cadorg** y uno a la matriz **char** que recibirá la cadena invertida en **cadr**. Por supuesto, la matriz señalada por **cadr** debe ser lo suficientemente grande para contener la cadena invertida más el terminador nulo. He aquí un ejemplo de cómo puede llamarse a **revsrtcpy()**:

```
char cad[5] = "abcd";
char inv[r];
invcad_copia(rev, cad);
```

Después de la llamada, **inv** contendrá los caracteres dcba y **cad** quedará sin modificación.

Ignore diferencias entre mayúsculas y minúsculas cuando compare cadenas terminadas en un carácter nulo

Componentes clave		
Encabezado	**Clases**	**Funciones**
<cctype>		int tolower(int *car*)

La función **strcmp()** es sensible a mayúsculas y minúsculas. Por tanto, las cadenas "prueba" y "Prueba" son diferentes a la comparación. Aunque una comparación sensible a mayúsculas y minúsculas suele ser lo que se necesita, hay ocasiones en que se requiere un método que no las diferencie. Por ejemplo, si está alfabetizando una lista de entradas para el índice de un libro, algunas de estas entradas podrían ser nombres propios, como los de una persona. A pesar de las diferencias entre mayúsculas y minúsculas, tal vez quiera que se preserve el orden alfabético. Por ejemplo, querrá que "Stroustrup" aparezca después de "clase". El problema es que las minúsculas están representadas por valores que son 32 más grandes que las mayúsculas. Por tanto, al realizar una comparación sensible a mayúsculas y minúsculas entre "Stroustrup" y "clase", la segunda aparece antes que la primera. Para resolver este problema, debe usar una función de comparación que ignore las diferencias entre mayúsculas y minúsculas. En esta solución se muestra una manera de hacerlo.

Paso a paso

Una manera de ignorar las diferencias entre mayúsculas y minúsculas cuando se comparan cadenas terminadas en un carácter nulo es crear su propia versión de la función **strcmp()**. Es muy fácil de hacer, como se muestra. La clave está en convertir cada conjunto de caracteres a mayúsculas

o minúsculas iguales y luego compararlas. En esta solución se convierten todos los caracteres a minúsculas utilizando la función **tolower()**, pero también funcionaría la conversión a mayúsculas.

1. Cree una función llamada **strcmp_ign_mayus()** que tenga este prototipo:

   ```
   int strcmp_ign_mayus(const char *cad1, const char *cad1);
   ```

2. Dentro de **strcmp_ign_mayus()**, compare cada carácter correspondiente en las dos cadenas. Para ello, configure un bucle que itere, siempre y cuando no se haya alcanzado el terminador de carácter nulo de una de las cadenas.
3. Dentro del bucle, convierta primero cada carácter a minúsculas, al llamar a **tolower()**. Luego compare los dos caracteres. Siga comparando caracteres hasta que se alcance el final de una de las cadenas, o cuando los dos caracteres sean diferentes. Observe que **tolower()** requiere el encabezado **<cctype>**.
4. Cuando el bucle se detiene, devuelve el resultado de restar el último carácter comparado de la segunda cadena al último carácter comparado de la primera cadena. Esto causa que la función devuelva menos de cero si *cad1* es menor que *cad2*, cero si los dos son iguales (en este caso, la terminación de carácter nulo de *cad2* se resta de la de *cad1*) o mayor que cero si *cad1* es mayor que *cad2*.

Análisis

La función estándar **tolower()** se definió originalmente en C y tiene soporte en C++ de dos maneras distintas. La versión usada aquí está declarada dentro del encabezado **<cctype>**. Convierte mayúsculas en minúsculas con base en un conjunto de caracteres definido por la configuración regional. Se muestra aquí:

int tolower(int *car*)

Devuelve el equivalente en minúsculas de *car*, que debe ser un valor de 8 bites. Los caracteres no alfabéticos se devuelven sin cambio.

Para comparar dos cadenas terminadas en un carácter nulo, independientemente de las diferencias entre mayúsculas y minúsculas, debe comparar los caracteres correspondientes en la cadena después de normalizarlos a mayúsculas o minúsculas. En la solución, los caracteres están convertidos en minúsculas. He aquí un ejemplo de un bucle que compara caracteres en dos cadenas pero ignora las diferencias entre mayúsculas y minúsculas:

```
while(*cad1 && *cad2) {
  if(tolower(*cad1) != tolower(*cad2))
    break;

  ++cad1;
  ++cad2;
}
```

Observe que el bucle se detendrá cuando se alcance el final de cualquier cadena o cuando se encuentre una diferencia.

Cuando el bucle termine, debe devolver un valor que indique el resultado de la comparación. Esto es fácil de hacer. Simplemente devuelve el resultado de restar el último carácter señalado por **cad2** al último carácter señalado por **cad1**, como se muestra aquí:

```
return tolower(*cad1) - tolower(*cad2);
```

Esto devuelve cero si se ha encontrado el terminador nulo de ambas cadenas, lo que indica igualdad. De otra manera, si el carácter señalado por **cad1** es menor que el señalado por **cad2**, se devuelve un valor negativo, lo que indica que la primera cadena es menor que la segunda. Si el carácter al que señala **cad1** es mayor que el señalado por **cad2**, se devuelve un valor positivo, lo que indica que la primera cadena es mayor que la segunda. Por tanto, produce el mismo resultado que **strcmp()**, pero de una manera no sensible a mayúsculas y minúsculas.

Ejemplo

Para unir todo, he aquí una manera de implementar una función de comparación de cadena no sensible a mayúsculas y minúsculas llamada **strcmp_ign_mayus()**:

```
// Una función de comparación simple de cadenas que ignora diferencias entre
// mayúsculas y minúsculas.
int strcmp_ign_mayus(const char *cad1, const char *cad2) {

  while(*cad1 && *cad2) {
    if(tolower(*cad1) != tolower(*cad2))
      break;

    ++cad1;
    ++cad2;
  }

  return tolower(*cad1) - tolower(*cad2);
}
```

El siguiente programa pone en funcionamiento a **strcmp_ign_mayus()**:

```
// Ignora la diferencia entre mayúsculas y minúsculas al comparar la cadena.
#include <iostream>
#include <cctype>

using namespace std;

int strcmp_ign_mayus(const char *str1, const char *cad2);
void mostrarresultado(const char *cad1, const char *cad2, int resultado);

int main() {
  char cadA[]= "pruebA";
  char cadB[] = "Prueba";
  char cadC[] = "pruebas";
  char cadD[] = "pre";

  int resultado;

  cout << "Las cadenas son: " << endl;
  cout << "cadA: " << cadA << endl;
  cout << "cadB: " << cadB << endl;
  cout << "cadC: " << cadC << endl;
  cout << "cadD: " << cadD << "\n\n";
```

```
  // Compara las cadenas ignorando mayúsculas y minúsculas.
  resultado = strcmp_ign_mayus(cadA, cadB);
  mostrarresultado(cadA, cadB, resultado);

  resultado = strcmp_ign_mayus(cadA, cadC);
  mostrarresultado(cadA, cadC, resultado);

  resultado = strcmp_ign_mayus(cadA, cadD);
  mostrarresultado(cadA, cadD, resultado);

  resultado = strcmp_ign_mayus(cadD, cadA);
  mostrarresultado(cadD, cadA, resultado);

  return 0;
}

// Una función de comparación simple de cadenas que ignora diferencias entre
// mayúsculas y minúsculas.
int strcmp_ign_mayus(const char *cad1, const char *cad2) {

  while(*cad1 && *cad2) {
    if(tolower(*cad1) != tolower(*cad2))
      break;

    ++cad1;
    ++cad2;
  }

  return tolower(*cad1) - tolower(*cad2);
}

void mostrarresultado(const char *cad1, const char *cad2, int resultado) {
  cout << cad1 << " is ";

  if(!resultado)
    cout << "igual a ";
  else if(resultado < 0)
    cout << "menor que ";
  else
    cout << "mayor que ";

  cout << cad2 << endl;
}
```

Aquí se muestra la salida:

```
Las cadenas son:
cadA: pruebA
cadB: Prueba
cadC: pruebas
cadD: pre
```

```
pruebA es igual a Prueba
pruebA es menor que pruebas
pruebA es mayor que pre
pre es menor que pruebA
```

Opciones

Como se explicó, la versión de **tolower()** declarada en **<cctype>** convierte caracteres con base en la configuración regional de idioma. Esto es lo que querrá casi siempre, de modo que es una buena (y conveniente) opción en casi todos los casos. Sin embargo, **tolower()** también se declara dentro de **<locale>**, que declara los miembros de la biblioteca de ubicación de C++. (La ubicación ayuda en la creación de código que puede internacionalizarse fácilmente.) He aquí esta versión de **tolower()**:

template <class charT> charT tolower(charT *car*, const locale &*loc*)

Esta versión de **tolower()** permite especificar una configuración diferente cuando se convierten las mayúsculas y minúsculas de una letra. Por ejemplo:

```
char car;
// . . .
locale loc("French");
cout << tolower(car, loc);
```

Esta llamada a **tolower()** usa la información de configuración regional y de idioma compatible con el francés.

Aunque no hay una ventaja en usarlo, también es posible convertir cada carácter en la cadena a mayúsculas (en lugar de minúsculas) para eliminar las diferencias entre mayúsculas y minúsculas. Esto se hace con la función **toupper()**, que se muestra aquí:

int toupper(int *car*)

Funciona de la misma manera que **tolower()**, excepto que convierte los caracteres en mayúsculas.

Cree una función de búsqueda y reemplazo para cadenas terminadas en un carácter nulo

Componentes clave		
Encabezado	**Clases**	**Funciones**
<ccstring>		char *strncpy(char **destino*, const char **origen*, int *cuenta*)
		void *memmove(void **destino*, const void **origen*, size_t *cuenta*)

Cuando se trabaja con cadenas, no es poco común que se necesite sustituir una subcadena con otra. Esta operación requiere dos pasos. En primer lugar, debe encontrar la subcadena que se

reemplazará y, en segundo lugar, debe reemplazarla con la nueva subcadena. A este proceso suele llamársele "búsqueda y reemplazo"; en esta solución se muestra una manera de realizar esto en cadenas terminadas en un carácter nulo.

Hay varias maneras de implementar una función de "búsqueda y reemplazo". Se usa un método en que el reemplazo tiene lugar en la cadena original, con lo que la modifica. En la sección *Opciones* se describen otros dos métodos para esta solución.

Paso a paso

Una manera de implementar una función de "búsqueda y reemplazo" para una cadena terminada en un carácter nulo incluye los siguientes pasos. Crea una función llamada **buscar_y_reemplazar()** que reemplaza la primera aparición de una subcadena con otra.

1. Cree una función llamada **buscar_y_reemplazar()**, que tenga este prototipo:

   ```
   bool buscar_y_reemplazar(char *cadorg, int longmax,
                                    const char *subcadant, const char *subcadnue)
   ```

   Se pasa un apuntador a la cadena original mediante **cadorg**. El número máximo de caracteres que **cadorg** puede tener se pasa en **longmax**. Un apuntador a la subcadena que se buscará se pasa mediante **subcadant,** y uno al reemplazo se pasa en **subcadnue**. La función devolverá un valor true si se hace una sustitución. Es decir, devuelve true si la cadena contenía originalmente por lo menos un caso de **subcadant**. Si no se realiza sustitución alguna, se devuelve false.
2. Busque una subcadena al llamar a **strstr()**. Devuelve un apuntador al principio de la primera subcadena de sustitución de un apuntador nulo, si no se encuentra una coincidencia.
3. Si se encuentra la subcadena, desplace los caracteres restantes en la cadena lo necesario para crear un "agujero" que sea exactamente del tamaño de la subcadena de reemplazo. Esto puede hacerse más fácil al llamar a **memmove()**.
4. Mediante **strncpy()**, copie la subcadena de reemplazo en el "agujero" en la cadena original.
5. Se devuelve el valor true si se hizo una sustitución y false si la cadena original queda sin cambio.

Análisis

Para encontrar una subcadena dentro de una cadena, use la función **strstr()** que se muestra aquí:

char *strstr(const char **cad1*, const char **cad2*)

Devuelve un apuntador al principio de la primera aparición de la cadena señalada por *cad2* en la cadena señalada por *cad1*. Si no se encuentra una coincidencia, se devuelve un apuntador nulo.

Desde el punto de vista conceptual, cuando se reemplaza una subcadena con otra, debe eliminarse la anterior e insertarse su reemplazo. En la práctica, no es necesario eliminar realmente la subcadena antigua. En cambio, simplemente puede sobreescribir la antigua con la nueva. Sin embargo, debe evitar que los caracteres restantes en la cadena se sobreescriban cuando la nueva subcadena sea más larga que la antigua. También debe asegurar que no quede un hueco cuando la nueva subcadena sea más corta que la antigua. Por tanto, a menos que la nueva subcadena sea del mismo tamaño que la antigua, necesitará subir o bajar los caracteres restantes en la cadena original para que cree un "hueco" en la cadena original que sea del mismo tamaño que la nueva. Una manera fácil de hacer esto consiste en usar **memmove()**, que se muestra a continuación:

void *memmove(void **destino*, const void **destino*, size_t *cuenta*)

Copia *cuenta* caracteres de la matriz señalada por *origen* en la matriz señalada por *destino*. Devuelve *destino*. La copia se realiza correctamente aunque las matrices se superpongan. Esto significa que puede usarse para subir o bajar caracteres en la misma matriz.

Después de que ha creado el "agujero" del tamaño apropiado en la cadena, puede copiar la nueva subcadena en él al llamar a **strncpy()**, como se muestra aquí:

char *strncpy(char **destino*, const char **origen*, size_t *cuenta*)

Esta función copia no más que *cuenta* caracteres de *origen* a *destino*. Si la cadena señalada por *origen* contiene menos que *cuenta* caracteres, se adjuntarán caracteres nulos al final de *destino* hasta que se hayan copiado *cuenta* caracteres. Sin embargo, si la cadena señalada por *origen* es más larga que *cuenta* caracteres, la cadena resultante no terminará en un carácter nulo. Devuelve *destino*. Si las dos cadenas se superponen, el comportamiento de **strncpy()** queda sin definir.

Haga que **buscar_y_reemplazar()** devuelva el valor true cuando tiene lugar una sustitución y false si no se encuentra la subcadena o si la cadena modificada excede la longitud máxima permisible de la cadena resultante.

Ejemplo

He aquí una manera de implementar la función **buscar_y_reemplazar ()**. Reemplaza la primera aparición de **subcadant** con **subcadnue**.

```
// Reemplaza la primera aparición de subcadant con subcadnue
// en la cadena señalada por cad. Esto significa que la función
// modifica la cadena señalada por cad.
//
// El tamaño máximo de la cadena resultante se pasa en longmax.
// Este valor debe ser menor que el tamaño de la matriz que
// contiene cad para evitar un desbordamiento de la matriz.
//
// Devuelve true si se hizo un reemplazo y falso, si no.
bool buscar_y_reemplazar(char *cad, int longmax,
                        const char *subcadant, const char *subcadnue) {

  // No permite que se sustituya el terminado de carácter nulo.
  if(!*subcadant) return false;

  // A continuación, revisa que la cadena resultante tenga una
  // longitud menor o igual al número máximo de caracteres permitido
  // en lo especificado por longmax. Si se excede el máximo, la
  // función termina al devolver false.
  int len = strlen(cad) - strlen(subcadant) + strlen(subcadnue);
  if(len > longmax) return false;

  // Ve si la subcadena especificada está en la cadena.
  char *p = strstr(cad, subcadant);

  // Si se encuentra la subcadena, se reemplaza con la nueva.
  if(p) {
```

```
    // Primero, se usa memmove() para mover el resto de la cadena
    // para que la nueva subcadena pueda reemplazar a la antigua.
    // En otras palabras, este paso aumenta o disminuye el tamaño
    // del "agujero" que llenará la nueva subcadena.
    memmove(p+strlen(subcadnue), p+strlen(subcadant),
            strlen(p)-strlen(subcadant)+1);
    // Ahora, copie la subcadena en cad.
    strncpy(p, subcadnue, strlen(subcadnue));

    return true;
  }

  // Devuelve false si no se hizo un reemplazo.
  return false;
}
```

Observe que la función no pondrá más de **longmax** caracteres en **cad**. El parámetro **longmax** se usa para evitar desbordamientos de matriz. Debe pasarle un valor que sea, como mínimo, uno menos que el tamaño de la matriz señalada por **cad**. Debe ser uno menos que el tamaño de la matriz porque debe abrir espacio para el terminador de carácter nulo.

En el siguiente programa se muestra la función **buscar_y_reemplazar()** en acción:

```
// Implementa "búsqueda y reemplazo" para cadena terminada en un carácter nulo.
#include <iostream>
#include <cstring>

 using namespace std;

bool buscar_y_reemplazar(char *cadorg, int longmax,
                        const char *subcadant, const char *subcadnue);

int main() {

  char cad[80] = "alfa beta gamma alfa beta gamma";

  cout << "Cadena original: " << cad << "\n\n";

  cout << "Primero, reemplaza todos los casos de alfa con omega.\n";

  // Reemplaza todas las apariciones de alfa con omega.
  while(buscar_y_reemplazar(cad, 79, "alfa", "omega"))
    cout << "Luego de un reemplazo: " << cad << endl;

  cout << "\nEnseguida, reemplaza todos los casos de gamma con zeta.\n";

  // Reemplaza todos los casos de gamma con zeta.
  while(buscar_y_reemplazar(cad, 79, "gamma", "zeta"))
   cout << "Luego de un reemplazo: " << cad << endl;

  cout << "\nAl final, elimina todas las apariciones de beta.\n";

  // Reemplaza todas las apariciones de beta con una cadena nula.
  // Esto se aplica al eliminar beta de la cadena.
  while(buscar_y_reemplazar(cad, 79, "beta", ""))
```

```
    cout << "Luego de un reemplazo: " << cad << endl;

  return 0;
}

// Reemplaza la primera aparición de subcadant con subcadnue
// en la cadena señalada por cad. Esto significa que la función
// modifica la cadena señalada por cad.
//
// El tamaño máximo de la cadena resultante se pasa en longmax.
// Este valor debe ser menor que el tamaño de la matriz que
// contiene cad para evitar un desbordamiento de la matriz.
//
// Devuelve true si se hizo un reemplazo y falso, si no.
bool buscar_y_reemplazar(char *cad, int longmax,
                        const char *subcadant, const char *subcadnue) {

  // No permite que se sustituya el terminado de carácter nulo.
  if(!*subcadant) return false;

  // A continuación, revisa que la cadena resultante tenga una
  // longitud menor o igual al número máximo de caracteres permitido
  // en lo especificado por longmax. Si se excede el máximo, la
  // función termina al devolver false.
  int len = strlen(cad) - strlen(subcadant) + strlen(subcadnue);
  if(len > longmax) return false;

   // Ve si la subcadena especificada está en la cadena.
  char *p = strstr(cad, subcadant);

  // Si se encuentra la subcadena, se reemplaza con la nueva.
  if(p) {

    // Primero, se usa memmove() para mover el resto de la cadena
    // para que la nueva subcadena pueda reemplazar a la antigua.
    // En otras palabras, este paso aumenta o disminuye el tamaño
    // del "agujero" que llenará la nueva subcadena.
    memmove(p+strlen(subcadnue), p+strlen(subcadant),
            strlen(p)-strlen(subcadant)+1);

    // Ahora, copie la subcadena en cad.
    strncpy(p, subcadnue, strlen(subcadnue));

    return true;
  }

  // Devuelve false si no se hizo un reemplazo.
  return false;
}
```

Aquí se muestra la salida:

```
Cadena original: alfa beta gamma alfa beta gamma

Primero, reemplaza todos los casos de alfa con omega.
```

```
Luego de un reemplazo: omega beta gamma alfa beta gamma
Luego de un reemplazo: omega beta gamma omega beta gamma

Enseguida, reemplaza todos los casos de gamma con zeta.
Luego de un reemplazo: omega beta zeta omega beta gamma
Luego de un reemplazo: omega beta zeta omega beta zeta

Al final, elimina todas las apariciones de beta.
Luego de un reemplazo: omega  zeta omega beta zeta
Luego de un reemplazo: omega  zeta omega  zeta
```

Opciones

Como está escrita, la función **buscar_y_reemplazar()** sustituye una subcadena dentro de la cadena original. Esto significa que ésta se modifica. Sin embargo, es posible emplear un método diferente en que la cadena original queda sin cambiar y la cadena sustituida se devuelve en otra matriz. Una manera de hacer esto consiste en pasar un apuntador a una cadena en que se copie el resultado. Esta técnica deja la cadena original sin cambio. Aquí se muestra esta opción:

```
// Esto reemplaza la primera aparición de subcadant con subcadnue.
// La cadena resultante se copia en la cadena pasada en
// cadresult. Esto significa que la cadena original queda sin
// cambio. La cadena resultante debe ser del largo suficiente para
// contener la cadena obtenida después de reemplazar subcadant con
// subcadnue. El número máximo de caracteres a copiar en cadresult
// se pasa en longmax. Devuelve true si se hizo un reemplazo,
// y false, de lo contrario.
bool buscar_y_reemplazar_copia(const char *cadorg, char *cadresult, int longmax,
                        const char *subcadant, const char *subcadnue) {

  // No permite que se sustituya el terminador de carácter nulo.
  if(!*subcadant) return false;

  // A continuación, revisa que la cadena resultante tenga una
  // longitud menor al número máximo de caracteres permitido
  // en lo especificado por longmax. Si se excede el máximo, la
  // función termina al devolver false.
  int len = strlen(cadorg) - strlen(subcadant) + strlen(subcadnue);
  if(len > longmax) return false;

  // Ve si la subcadena especificada está en la cadena.
  const char *p = strstr(cadorg, subcadant);

  // Si se encuentra la subcadena, se reemplaza con la nueva.
  if(p) {

    // Copia la primera parte de la cadena original.
    strncpy(cadresult, cadorg, p-cadorg);

    // Termina con un carácter nulo la primera parte de cadresult
```

```
    // para que operen en él las otras funciones de cadena.
    *(cadresult + (p-cadorg)) = '\0';

    // Sustituye la nueva subcadena.
    strcat(cadresult, subcadnue);

    // Agrega el resto de la cadena original,
    // sobre la subcadena que se reemplazó.
    strcat(cadresult, p+strlen(subcadant));

    return true;
  }

  // Devuelve false si no se hizo un reemplazo.
  return false;
}
```

Los comentarios dan una descripción "paso a paso" de la manera en que funciona **buscar_y_reemplazar_copia()**. He aquí un resumen. La función empieza por encontrar la primera aparición de **cadorg** de la subcadena pasada en **subcadant**. Luego copia la cadena original (**cadorg**) en la cadena resultante (**cadresult**) hasta el punto en que se encontró la subcadena. A continuación, copia la cadena de reemplazo en **cadresult**. Por último, copia el resto de **cadorg** en **cadresult**. Por tanto, al regresar, **cadresult** contiene una copia de **cadprg**, con la única diferencia de la sustitución de **subcadnue** por **subcadant**. Para evitar el desbordamiento de la matriz, **buscar_y_reemplazar_copia()** sólo copiará hasta **longmax** caracteres en **cadresult**. Por tanto, la matriz señalada por **cadresult** debe tener por lo menos **longmax**+1 caracteres de largo. El carácter extra deja espacio para el terminador de carácter nulo.

Otra opción útil en algunos casos consiste en hacer que la función **buscar_y_reemplazar()** asigne dinámicamente una nueva cadena que contenga la cadena resultante y devuelva un apuntador a ella. Este método ofrece una gran ventaja: no necesita preocuparse por el hecho de que se desborden los límites de la matriz porque puede asignar una matriz de tamaño apropiado. Esto significa que no es necesario que conozca el tamaño de la cadena resultante de antemano. La principal desventaja es que debe acordarse de eliminar la cadena asignada dinámicamente cuando ya no se necesite. He aquí una manera de implementar este método:

```
// Reemplaza la primera aparición de subcadant con subcadnue
// en cad. Devuelve un apuntador a una nueva cadena que contiene
// el resultado. La cadena señalada por cad queda sin cambio.
// Se asigna dinámicamente memoria para la nueva cadena y debe
// liberarse cuando ya no se necesite. Si no se hace una sustitución,
// se devuelve un apuntador nulo. Esta función lanza mala_asign si
// ocurre una falla en la asignación de memoria.
char *buscar_y_reemplazar_asign(const char *cad, const char *subcadant,
                                const char *subcadnue) throw(mala_asign) {

  // No permite que se sustituya el terminador de carácter nulo.
  if(!*subcadant) return 0;
```

```
  // Asigna una matriz con el largo suficiente para contener la cadena
  //resultante.
  int tam = strlen(cad) + strlen(subcadnue) - strlen(subcadant) + 1;
  char *resultado = new char[tam];

  const char *p = strstr(cad, subcadant);

  if(p) {

    // Copia primero la parte de la cadena original.
    strncpy(resultado, cad, p-cad);

    // Termina con un carácter nulo la primera parte del resultado
    // para que las otras funciones de la cadena operan en él.
    *(resultado+(p-cad)) = '\0';

    // Sustituye la nueva cadena.
    strcat(resultado, subcadnue);

    // Agrega el resto de la cadena original.
    strcat(resultado, p+strlen(subcadant));
  } else {
    delete [] resultado; // libera la memoria no utilizada.
    return 0;
  }

  return resultado;
}
```

Observe que **buscar_y_reemplazar_asign()** lanza **mala_asign()** si falla la asignación de la matriz temporal. Recuerde que la memoria es finita y que puede quedarse sin ella. Esto es especialmente cierto para los sistemas incrustados. Por tanto, el llamador de esta versión tal vez necesite manejar esta excepción. Por ejemplo, he aquí el marco conceptual básico que puede utilizar para llamar a **buscar_y_reemplazar_asign()**:

```
char *apt;
try {
  apt = buscar_y_reemplazar_asign(cad, ant, nue)
} catch(mala_asign excepción) {
  // Aquí se toma la acción apropiada.
}

if(apt) {
  // Usa la cadena...

  // Elimina la memoria cuando ya no se necesite.
  delete [] apt;
}
```

Ordene en categorías caracteres dentro de una cadena terminada en un carácter nulo

Componentes clave		
Encabezado	**Clases**	**Funciones**
<cctype>		int isalnum(int *car*)
		int isalpha(int *car*)
		int iscntrl(int *car*)
		int isdigit(int *car*)
		int isgraph(int *car*)
		int islower(int *car*)
		int isprint(int *car*)
		int ispunct(int *car*)
		int isspace(int *car*)
		int isupper(int *car*)
		int isxdigit(int *car*)

En ocasiones, querrá saber qué tipos de caracteres contiene una cadena. Por ejemplo, tal vez quiera eliminar todos los espacios en blanco (espacios, tabuladores y saltos de línea) de un archivo o el despliegue de caracteres que no se imprimen usando algún tipo de representación visual. Realizar estas tareas significa que puede ordenar los caracteres en tipos diferentes, como alfabético, control, dígitos, puntuación, etc. Por fortuna, C++ facilita mucho la realización de esto empleando una o más funciones estándar que determinan una categoría de caracteres.

Paso a paso

Las funciones de carácter facilitan el ordenamiento en categorías de un carácter. Incluye estos pasos:

1. Todas las funciones de ordenamiento de caracteres en categorías se declaran en **<cctype>.** Por tanto, deben incluirse en su programa.
2. Para determinar si un carácter es una letra o un dígito, llame a **int isalnum(int *car*)**.
3. Para determinar si un carácter es una letra, llame a **int isalpha(int *car*)**.
4. Para determinar si un carácter es un carácter de control, llame a **int iscntrl(int *car*)**.
5. Para determinar si un carácter es un dígito, llame a **int isdigit(int *car*)**.
6. Para determinar si un carácter es visible, llame a **int isgraph(int *car*)**.
7. Para determinar si un carácter es una letra minúscula, llame a **int islower(int *car*)**.
8. Para determinar si un carácter es imprimible, llame a **int isprint(int *car*)**.
9. Para determinar si un carácter es un signo de puntuación, llame a **int ispunct(int *car*)**.
10. Para determinar si un carácter es un espacio en blanco, llame a **int isspace(int *car*)**.
11. Para determinar si un carácter es una letra mayúscula, llame a **int isupper(int *car*)**.
12. Para determinar si un carácter es un dígito hexadecimal, llame a **int isxdigit(int *car*)**.

Análisis

Las funciones de ordenamiento de caracteres en categorías se definieron originalmente en C y tienen soporte en C++ en un par de maneras diferentes. Las versiones empleadas aquí se declaran dentro del encabezado **<cctype>**. Todos ordenan en categorías los caracteres con base en la configuración local y de idioma.

Todas las funciones **is. . .** se desempeñan exactamente de la misma manera. Cada una se describe brevemente aquí:

int isalnum(int *car*)	Devuelve un valor diferente de cero si *car* es una letra o un dígito, y cero de otra manera.
int isalpha(int *car*)	Devuelve un valor diferente de cero si *car* es una letra, y cero de otra manera.
int iscntrl(int *car*)	Devuelve un valor diferente de cero si *car* es un carácter de control, y cero de otra manera.
int isdigit(int *car*)	Devuelve un valor diferente de cero si *car* es un dígito, y cero de otra manera.
int isgraph(int *car*)	Devuelve un valor diferente de cero si *car* es un carácter imprimible diferente un espacio, y cero de otra manera.
int islower(int *car*)	Devuelve un valor diferente de cero si *car* es una letra minúscula, y cero de otra manera.
int isprint(int *car*)	Devuelve un valor diferente de cero si *car* es imprimible (incluido un espacio), y cero de otra manera.
int ispunct(int *car*)	Devuelve un valor diferente de cero si *car* es un signo de puntuación, y cero de otra manera.
int isspace(int *car*)	Devuelve un valor diferente de cero si *car* es un espacio en blanco, y cero de otra manera.
int isupper(int *car*)	Devuelve un valor diferente de cero si *car* es una letra mayúscula, y cero de otra manera.
int isxdigit(int *car*)	Devuelve un valor diferente de cero si *car* es un dígito hexadecimal (0-9, A-F o a-f), y cero de otra manera.

Casi todas las funciones se explican por sí solas. Sin embargo, observe que la función **ispunct()** devuelve un valor true para cualquier carácter que sea un signo de puntuación. Esto se define como cualquier carácter que no sea una letra, un dígito o un espacio. Por tanto, operadores como + y / se ordenan en categorías como signos de puntuación.

Ejemplo

En el siguiente ejemplo se muestran en acción las funciones **isalpha()**, **isdigit()**, **isspace()** e **ispunct()**. Se usan para contar el número de letras, espacios y signos de puntuación contenidos dentro de una cadena.

```
// Cuenta espacios, signos de puntuación, dígitos y letras.
#include <iostream>
#include <cctype>

using namespace std;

int main() {

  const char *cad = "Tengo 30 manzanas y 12 peras. \u00a8Tienes algo?";
  int letras = 0, espacios = 0, punt = 0, dígitos = 0;

  cout << cad << endl;
```

```
  while(*cad) {
    if(isalpha(*cad)) ++letras;
    else if(isspace(*cad)) ++espacios;
    else if(ispunct(*cad)) ++punt;
    else if(isdigit(*cad)) ++dígitos;

    ++cad;
  }

  cout << "Letras: " << letras << endl;
  cout << "D\u00a1gitos: " << digitos << endl;
  cout << "Espacios: " << espacios << endl;
  cout << "Signos de puntuaci\u00a2n: " << punt << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Tengo 30 manzanas y 12 peras. ¿Tienes algo?
Letras: 29
Dígitos: 4
Espacios: 7
Signos de puntuación: 2
```

Ejemplo adicional: conteo de palabras

Hay una aplicación bien conocida en que se usan las funciones de ordenamiento de caracteres en categorías: una utilería de conteo de palabras. Como resultado, un programa para este fin es el ejemplo quintaesencial para funciones como **isalpha()** e **ispunct()**. En el siguiente ejemplo se crea una versión muy simple de la utilería de conteo de palabras. El conteo real lo maneja la función **contarpalabras()**. Se pasa un apuntador a una cadena. Luego cuenta las palabras, líneas, espacios y signos de puntuación en la cadena y devuelve el resultado.

Esta versión de **contarpalabras()** usa una estrategia muy simple: sólo cuenta palabras completas que están integradas exclusivamente por letras. Esto significa que una palabra con guiones cuenta como dos palabras separadas. Como resultado, la sección "terminada en un carácter nulo" cuenta como dos palabras. Más aún, una palabra no debe contener cualquier dígito. Por ejemplo, la secuencia "probando123probando" contará como dos palabras. La función **contarpalabras(),** no obstante, permite que un carácter diferente de una letra esté en una palabra: el apóstrofo. Esto permite el uso de posesivos en inglés (como Tom's) y contracciones (como it's). Sin embargo, la función **contarpalabras()**.

```
// Cuenta palabras, líneas, espacios y signos de puntuación.
#include <iostream>
#include <cctype>

using namespace std;

// Una estructura que contiene las estadísticas de conteo de palabras.
struct cp {
  int palabras;
  int espacios;
  int punt;
  int lineas;
```

```
  cp() {
    palabras = punt = espacios = líneas = 0;
  }
};

cp contarpalabras(const char *cad);

int main() {

  const char *prueba = "Al proporcionar una clase de cadena y dar "
                       "soporte a cadenas terminadas-en-nulo,\nC++ "
                       "ofrece un entorno rico para tareas intensas en "
                       "cadenas,\incluido el uso de signos como el de Mario's House.";


  cout << "Dada la frase: " << "\n\n";
  cout << prueba << endl;
  cp cpal = contarpalabras(prueba);

  cout << "\nPalabras: " << cpal.palabras << endl;
  cout << "Espacios: " << cpal.espacios << endl;
  cout << "L\u00a1neas: " << cpal.lineas << endl;
  cout << "Signos de puntuaci\u00a2n: " << cpal.punt << endl;

  return 0;
}

// Una función muy simple de "conteo de palabras".
// Cuenta las palabras, espacios y signos de puntuación en
// una cadena y devuelve el resultado en una estructura cp.
cp contarpalabras(const char *cad) {
  cp datos;

  // Si la cadena no es nula, entonces contiene por lo menos una línea.
  if(*cad) ++datos.líneas;

  while(*cad) {

    // Revisa una palabra.
    if(isalpha(*cad)) {
      // Inicia  la búsqueda de palabras. Ahora busca el final.
      // de las palabras. Permite apóstrofes en las palabras.".
      while(isalpha(*cad) || *cad == '\'') {
        if(*cad == '\'') ++datos.punt;
        ++cad;
      }
      datos.palabras++;
    }
    else {
      // Cuenta signos de puntuación, espacios (incluidos saltos de página) y
líneas.
      if(ispunct(*cad)) ++datos.punt;

      else if(isspace(*cad)) {
```

```
        ++datos.espacios;
        // Si hay algún carácter después del salto de línea, aumenta
        // el contador de líneas.
        if(*cad == '\n' && *(cad+1)) ++datos.lineas;
      }
      ++cad;
    }
  }

  return datos;
}
```

Aquí se muestra la salida:

```
Dada la frase:

Al proporcionar una clase de cadena y dar soporte a cadenas terminadas-en-nulo,
C++ ofrece un entorno rico para tareas intensas en cadenas,
incluido el uso de signos como el de Mario's House.

Palabras: 34
Espacios: 31
Líneas: 3
Signos de puntuación: 8
```

Hay un par de temas de interés en este programa. En primer lugar, observe que la función **contarpalabras()** devuelve los resultados en un objeto de tipo **cp**, que es una **struct**. Se usó una **struct** en lugar de una **class** porque **cp** es, en esencia, un objeto de sólo datos. Aunque **cp** no contiene un constructor predeterminado (que realiza una inicialización simple), no define funciones miembro o constructores parametrizados. Por tanto, **struct** cumple mejor su propósito (que es contener datos) que **class**. En general, es preferible usar **class** cuando hay funciones de miembros. Se prefiere usar **struct** con objetos que simplemente hospedan datos. Por supuesto, en C++, ambos crean un tipo de clase y no hay una regla inmutable al respecto.

En segundo lugar, el conteo de líneas aumenta cuando se encuentra un carácter de nueva línea sólo si no va seguido inmediatamente después por un carácter de terminación nulo. Esta comprobación se maneja con esta línea:

```
if(*cad == '\n' && *(cad+1)) ++datos.lineas;
```

En esencia, esto asegura que el número de líneas de texto que se vería es igual a la cuenta de líneas devuelta por la función. Esto evita que una línea final completamente vacía se cuente como una línea. Por supuesto, la línea aún puede aparecer en blanco si todo lo que contiene son espacios.

Opciones

Como se mencionó, las funciones de ordenamiento en categorías definidas en **<cctype>** se relacionan con la configuración regional y de idioma. Además, versiones de estas funciones también están soportadas por **<locale>** y permiten especificar una configuración.

Convierta en fichas una cadena terminada en un carácter nulo

Componentes clave		
Encabezado	**Clases**	**Funciones**
<cstring>		char *strtok(char **cad*, const char **delimitadores*);

La conversión en fichas de una cadena es una tarea de programación que casi todo programador enfrentará en un momento u otro. *Convertir en fichas* es el proceso de reducir una cadena a sus partes individuales, a las que se les denomina *fichas* (o token). Por tanto, una ficha representa el elemento indivisible más pequeño que puede extraerse de una cadena y que signifique algo.

Por supuesto, lo que constituye una ficha depende de cada tipo de entrada que se está procesando y su propósito. Por ejemplo, si quiere obtener las palabras en una frase, entonces una ficha es un conjunto de caracteres rodeados por espacios en blanco o signos de puntuación. Por ejemplo, dada la frase:

Yo prefiero manzanas, peras y uvas.

Las fichas individuales son:

Yo	prefiero	manzanas
peras	y	uvas

Cada ficha está delimitada por el espacio en blanco o el signo de puntuación que separa a una de otra. Cuando se convierte una cadena en fichas que contiene una lista de pares clave/valor organizados de esta manera:

Clave=valor, clave=valor, clave=valor, ...

Las fichas son la *clave* y el *valor*. El signo = y la coma son separadores que delimitan las fichas. Por ejemplo, dado

precio=10.15, cantidad=4

Las fichas son

precio	10.15	cantidad	4

Lo importante es que lo que constituye una ficha cambiará, dependiendo de la circunstancia. Sin embargo, el proceso general de convertir una cadena en fichas es el mismo en todos los casos.

Debido a que convertir una cadena en fichas es una tarea importante y común, C++ proporciona soporte integrado mediante la función **strtok()**. En la solución se muestra cómo usarla.

Paso a paso

Para usar **strtok()** para convertir una cadena en fichas se requieren estos pasos:

1. Cree una cadena que contenga los caracteres que separan una ficha de otra. Hay delimitadores de fichas.
2. Para obtener la primera ficha en la cadena, llame a **strtok()** con un apuntador a la cadena que se convertirá en ficha y uno a la cadena que contiene los delimitadores.
3. Para obtener las fichas restantes en la cadena, siga llamando a **strtok()**. Sin embargo, pase un apuntador nulo para el primer argumento. Puede cambiar los delimitadores, de acuerdo con lo necesario.
4. Cuando **strtok()** devuelve null, la cadena se ha convertido por completo en fichas.

Análisis

La función **strtok()** tiene el siguiente prototipo:

char *strtok(char **cad*, const char **delimitadores*)

Un apuntador a la cadena desde la que se obtendrán una o más fichas se pasa en *cad*. Un apuntador a la cadena que contiene los caracteres que delimitan una ficha se pasan en *delimitadores*. Por tanto, *delimitadores* contiene los caracteres que dividen una ficha de otra. Un apuntador nulo se devuelve si ya no hay más fichas en *cad*. De otra manera, se devuelve un apuntador a una cadena que contiene la siguiente ficha.

La conversión de cadenas en fichas es un proceso de dos pasos. La primera llamada a **strtok()** pasa un apuntador a la cadena que se habrá de convertir en ficha. Cada llamada posterior a **strtok()** pasa un apuntador nulo a *cad*. Esto causa que **strtok()** siga convirtiendo en fichas la cadena desde el punto en que se encontró la ficha anterior. Cuando ya no se encuentran más fichas, se devuelve un apuntador nulo.

Un aspecto útil de **strtok()** es que puede cambiar los delimitadores necesarios durante el proceso de conversión en fichas. Por ejemplo, tome en consideración la cadena que contiene pares clave/valor organizados de la manera siguiente:

cuenta = 10, max = 99, menú Inicio = 12, nombre = "Tom Jones, jr.", ...

Para leer casi todas las claves y los valores de esta cadena, puede utilizarse el siguiente conjunto delimitador:

"=,"

Sin embargo, para leer una cadena entre comillas que incluye cualquier carácter, incluidas comas, se necesita este delimitador:

"\""

Debido a que **strtok()** le permite cambiar conjuntos de delimitadores "al vuelo", puede especificar los delimitadores que son necesarios en cualquier momento. Esta técnica se ilustra en el siguiente ejemplo.

Ejemplo

Se muestra cómo usar **strtok()** para convertir en fichas una cadena terminada en un carácter nulo:

```
// Demuestra strtok().
#include <iostream>
#include <cstring>

using namespace std;

int main() {

  // Primero, usa strtok() para convertir una frase en fichas.

  // Crea una cadena de delimitadores para frases simple.
  char delims[] = ".,\u00a8 ?;!";

  char cad[] = "Yo prefiero manzanas, peras y uvas. \u00a8Y t\u00a3?";

  char *ficha;

  cout << "Obtiene las palabras de una frase.\n";

  // Pasa la cadena que se convertirá en fichas y obtiene la primera ficha.
  ficha = strtok(cad, delims);

  // Obtiene todas las fichas restantes.
  while(ficha) {
    cout << ficha << endl;

    // Cada llamada posterior a strtok() pasa NULL
    // para el primer argumento.
    ficha  = strtok(NULL, delims);
  }

  // Ahora, usa strtok() para extraer claves y valores almacenados
  // en pares clave/valor dentro de una cadena.
  char parescv[] = "cuenta=10, nombre=\"Tom Jones, jr.\", max=100, min=0.01";

  // Crea una lista de delimitadores para pares clave/valor.
  char delimscv[] = " =,";

  cout << "\nConvierte en fichas valores clave/valor.\n";

   // Obtiene la primera clave.
  ficha = strtok(parescv, delimscv);

  // Obtiene las fichas restantes.
  while(ficha) {
    cout << "Clave: " << ficha << " ";

    // Obtiene un valor.

    // Primero, si la clave es nombre, el valor será
    // una cadena entre comillas.
    if(!strcmp("nombre", ficha)) {
      // Observe que esta llamada usa sólo comillas como delimitador. Esto le
      // permite leer una cadena entre comillas que incluye cualquier carácter.
```

```
      ficha = strtok(NULL, "\"");
    }
    else {
      // De otra manera, lee un valor simple.
      ficha  = strtok(NULL, delimscv);
    }
    cout << "Valor: " << ficha << endl;

    // Obtiene la siguiente clave.
    ficha = strtok(NULL, delimscv);
  }

  return 0;
}
```

Aquí se muestra la salida:

```
Obtiene las palabras de una frase.
Yo
prefiero
manzanas
peras
y
uvas
Y
tú

Convierte en fichas valores clave/valor.
Clave: cuenta Valor: 10
Clave: nombre Valor: Tom Jones, jr.
Clave: max Valor: 100
Clave: min Valor: 0.01
```

Preste especial atención a la manera en que se leen los pares clave/valor. Los delimitadores usados para leer un valor simple difieren de los usados para leer una cadena entre comillas. Más aún, los delimitadores se cambian durante el proceso de conversión en fichas. Como ya se explicó, cuando se convierte una cadena en fichas, puede cambiar el conjunto de delimitadores a medida que lo necesite.

Opciones

Aunque el uso de **strtok()** es simple y muy efectivo cuando se aplica en situaciones para las que resulta adecuado, su uso está inherentemente limitado. El principal problema es que **strtok()** convierte en fichas una cadena basada en un conjunto de delimitadores, y una vez que se ha encontrado uno, se pierde. Esto dificulta el uso de **strtok()** para convertir en fichas una cadena en que los delimitadores podrían también ser fichas. Por ejemplo, considere la siguiente instrucción simple de C++:

x = cuenta+12;

Para analizar esta instrucción, el signo + debe manejarse como un delimitador que termina **cuenta** y como una ficha que indica la suma. El problema es que no hay una manera fácil de hacer esto empleando **strtok()**. Para obtener **cuenta**, el + debe estar en el conjunto de delimitadores. Sin embargo, una vez que el + se ha encontrado, se consume. Por tanto, tampoco puede leerse como una ficha. Un segundo

problema con **strtok()** es que resulta difícil detectar los errores en el formato de la cadena que se está convirtiendo en fichas (por lo menos hasta que se alcanza prematuramente el final de la cadena).

Debido al problema de aplicar **strtok()** a un amplio rango de casos, suelen usarse otros métodos para convertir en fichas. Uno de éstos consiste en escribir su propia función "para obtener fichas". Esto le da control completo sobre el proceso de conversión en fichas y le permite regresar fácilmente fichas basadas en el contexto más que en delimitadores. Aquí se muestra un ejemplo simple de este método. A la función personalizada para obtener fichas se le denomina **obtenerfichas()**. Se convierte en fichas una cadena en los siguientes tipos de fichas:

- Cadenas alfanuméricas, como cuenta, índice27 o WordPad.
- Números enteros sin signo, como 2, 99 o 0.
- Signos de puntuación, entre ellos operadores, como + y /.

Por tanto, **obtenerfichas()** puede usarse para convertir en fichas expresiones muy simples, como

x = cuenta+12

o

while(x<9)x = x –w;

La función **obtenerfichas()** se usa de manera parecida a **strtok()**. En la primera llamada, se pasa un apuntador a la cadena que habrá de convertirse en fichas. En llamadas posteriores, se pasa un apuntador nulo. Devuelve un apuntador cuando no hay más fichas. Para convertir en fichas una nueva cadena, simplemente empiece el proceso al pasar un apuntador a la nueva cadena. Aquí se muestra la función **obtenerfichas()** simple, junto con una función **main()** para demostrar su uso:

```
// Demuestra una función obtenerficha() personalizada que puede
// devolver las fichas incluidas en expresiones muy simples.
#include <iostream>
#include <cstring>
#include <cctype>

using namespace std;

const char *obtenerficha(const char *cad);

int main() {
  char ejemploA[] = "max=12+3/89; cuenta27 = 19*(min+piso);";
  char ejemploB[] = "while(i < max) i = contador * 2;";
  const char *fic;

  // Convierte en fichas la primera cadena.
  fic = obtenerficha(ejemploA);
  cout << "Fichas que se encuentran en: " << ejemploA << endl;
  while(fic) {
    cout << fic << endl;
    fic  = obtenerficha(NULL);
  }
  cout << "\n\n";
```

```
  // Reinicia obtenerficha() al pasar la segunda cadena.
  fic = obtenerficha(ejemploB);
  cout << "Fichas que se encuentran en: " << ejemploB << endl;
  while(fic) {
    cout << fic << endl;
    fic  = obtenerficha(NULL);
  }

  return 0;
}

// Una función obtenerficha() personalizada muy simple. Las fichas están
// formadas por cadenas alfanuméricas, números y signos de puntuación de
// un solo carácter. Aunque esta función es muy limitada, demuestra el
// marco conceptual básico que puede expandirse y mejorarse para obtener
// otros tipos de fichas.
//
// En la primera llamada, pasa un apuntador a la cadena que se convertirá.
// en fichas. En llamadas posteriores, pasa un apuntador nulo. Devuelve
// un apuntador a la ficha actual, o un apuntador nulo si es que no hay
// más fichas.
#define TAM_MAX_FICHA 128
const char *obtenerficha(const char *cad) {
  static char ficha[TAM_MAX_FICHA+1];
  static const char *apt;
  int cuenta; // contiene la cuenta actual de caracteres
  char *aptficha;

  if(cad) {
    apt = cad;
  }

  aptficha = ficha;
  cuenta = 0;

  while(isspace(*apt)) apt++;

  if(isalpha(*apt)) {
    while(isalpha(*apt) || isdigit(*apt)) {
      *aptficha++ = *apt++;
      ++cuenta;
      if(cuenta == TAM_MAX_FICHA) break;
    }
  } else if(isdigit(*apt)) {
    while(isdigit(*apt)) {
      *aptficha++ = *apt++;
      ++cuenta;
      if(cuenta == TAM_MAX_FICHA) break;
    }
  } else if(ispunct(*apt)) {
    *aptficha++ = *apt++;
  } else return NULL;

  // Null termina la ficha.
  *aptficha = '\0';

  return ficha;
}
```

Aquí se muestra la salida del programa:

```
Fichas que se encuentran en: max=12+3/89; cuenta27 = 19*(min+piso);
max
=
12
+
3
/
89
;
cuenta27
=
19
*
(
min
+
piso
)
;


Fichas que se encuentran en: while(i < max) i = contador * 2;
while
(
i
<
max
)
i
=
contador
*
2
;
```

La operación de **obtenerfichas()** es muy sencilla. Simplemente examina el siguiente carácter de la cadena de entrada y luego lee el tipo de ficha con que inicia ese tipo de carácter. Por ejemplo, si la ficha es una letra, entonces **obtenerfichas()** lee una ficha alfanumérica. Si el siguiente carácter es un dígito, entonces lee un entero. Si es un signo de puntuación, entonces la ficha contiene ese carácter. Observe que **obtenerfichas()** no deja que la longitud de una ficha exceda la longitud máxima de la ficha, especificada en **TAM_MAX_FICHA**. También observe que **obtenerfichas()** *no* modifica la cadena de entrada. Esto difiere de **strtok()**, que sí la modifica. Por último, observe que el apuntador devuelto por **obtenerfichas()** es **const**. Esto significa que puede usarse para modificar la matriz estática **ficha**. Por último, aunque **obtenerfichas()** es muy simple, puede adaptarse y mejorarse fácilmente para adecuarse a otras situaciones, más complejas.

Realice operaciones básicas en objetos de string

Componentes clave		
Encabezado	**Clases**	**Funciones**
<string>	string	size_type capacity() const string &erase(size_type *ind* = 0, size_type *long* = npos) string &insert(size_type *ind*, const string &*cad* size_type max_size() const char &operator[](size_type *ind*) string &operator=(const string &*cad*) void push_back(const char *car*) void reserve(size_type *num* = 0) size_type size() const; string substr(size_type *ind* = 0, size_type *long* = npos) const
<string		string operator+(const string %*izqarr*, const string %*derarr*) string operator==(const string %*izqarr*, const string %*derarr*) bool operator<=(const string %*izqarr*, const string %*derarr*) bool operator>(const string %*izqarr*, const string %*derarr*)

Como se explicó al principio de este capítulo, C++ proporciona dos maneras de manejar cadenas. La primera es la cadena terminada en un carácter nulo (también llamada cadena C). Ésta se heredó de C y aún se usa ampliamente en programación con C++. También es el tipo de cadena usado en las soluciones anteriores. El segundo tipo de cadena es un objeto de la clase de plantillas **basic_string**. La cual está definida por C++ y es parte de la biblioteca de clases estándar de C++. En el resto de las soluciones de este capítulo se utiliza **basic_string**.

Las cadenas de tipo **basic_string** tienen varias ventajas sobre las terminadas en un carácter nulo. He aquí algunas de las más importantes:

- **basic_string** define un tipo de datos. (Recuerde que una cadena terminada en un carácter nulo es simplemente una convención.)
- **basic_string** encapsula la secuencia de caracteres que forma la cadena, con lo que evita operaciones inapropiadas. Cuando se usa **basic_string**, no es posible generar un desbordamiento de matriz, por ejemplo.
- Los objetos de **basic_string** son dinámicos. Crecen a medida que se necesitan para acomodarse al tamaño de la cadena que se está incluyendo. Por tanto, no es necesario saber de antemano el largo de la cadena.
- **basic_string** define operadores que manipulan cadenas. Esto simplifica muchos tipos de manejo de cadenas.

- **basic_string** define un conjunto completo de funciones de miembro que simplemente funcionan con cadenas. Pocas veces tiene que escribir su propia función para realizar alguna manipulación de cadenas.

Hay dos especializaciones integradas de **basic_string**: **string** (que es para caracteres de tipo **char**) y **wstring** (que es para caracteres ampliados). Por conveniencia, todas las soluciones de este libro usan **string**, pero casi toda la información es aplicable a cualquier tipo de **basic_string**.

En esta solución se demuestran varias de las operaciones básicas que pueden aplicarse a objetos de tipo **string**. Se muestra cómo construir una **string**. Luego se presentan varios de sus operadores y funciones miembro. También se demuestra cómo ajustan su tamaño en tiempo de ejecución los objetos **string** para acomodar un aumento en el tamaño de la secuencia de caracteres.

Paso a paso

Para realizar las operaciones básicas de **string** se requieren estos pasos:

1. La clase **string** está declarada dentro del encabezado **<string>**. Por tanto, debe incluirse éste en cualquier programa que utilice **string**.
2. Un objeto **string** se crea al usar uno de sus constructores. En esta solución se demuestran tres. El primero crea una **string** vacía, el segundo crea una inicializada por una literal **string**, y el tercero crea una que se inicializa con otra.
3. Para obtener la longitud de la cadena más larga posible, llame a **max_size()**.
4. Para asignar una cadena a otra, use el operador **=**.
5. Para unir dos objetos **string**, use el operador **+**.
6. Para comparar lexicográficamente dos objetos **string**, use los operadores relacionales, como **>** o **==**.
7. Para obtener una referencia a un carácter en un índice especificado, use el operador de indización **[]**.
8. Para obtener el número de caracteres contenido actualmente por **string**, llame a **size()**.
9. Para obtener la capacidad actual de **string**, llame a **capacity()**.
10. Para especificar una capacidad, llame a **reserve()**.
11. Para eliminar todos los caracteres o parte de ellos en una **string**, llame a **erase()**.
12. Para agregar un carácter al final de una cadena, llame a **push_back()**.
13. Para obtener una subcadena, llame a **substr()**.

Análisis

La clase **string** define varios constructores. Aquí se muestran los usados en esta solución:

```
explicit string(const Allocator &asign = Allocator())
string(const char *cad, const Allocator &asign = Allocator())
string(const string &cad, size_type ind = 0,
    size_type long = npos Allocator &asign = Allocator())
```

El primer constructor crea una cadena vacía. El segundo, una que es inicializada por una cadena terminada en un carácter nulo señalada por *cad*. El tercero, una que es inicializada por una subcadena de *cad* que empieza en *ind* y se ejecuta por *long* caracteres. Aunque parecen un poco intimidantes, son fáciles de usar. Por lo general, el asignador (que controla la manera en que se asigna

la memoria) está permitido como opción predeterminada. Esto significa que, por lo general, no querrá especificar un asignador cuando crea una **string**. Por ejemplo, lo siguiente crea una cadena vacía y una inicializada con una literal de cadena:

```
string micad; // cadena vacía
string micad2("Hola"); //cadena inicializada con la secuencia Hola
```

En el tercer constructor, suelen usarse las opciones predeterminadas para *ind* y *long*, lo que significa que la cadena contiene una copia completa de *cad*.

Aunque los objetos **string** son dinámicos, y crecen de acuerdo con las necesidades en tiempo de ejecución, una cadena aún puede tener una longitud máxima. Aunque este máximo suele ser demasiado, tal vez sea útil conocerlo en algunos casos. Para obtener la longitud de cadena máxima, llame a **max_size()**, que se muestra aquí:

```
size_type max_size() const
```

Devuelve la longitud de la cadena más larga posible.

Puede asignar una **string** a otra al usar el operador **=**. Éste se ha implementado como una función miembro. Tiene varias formas. He aquí una usada en esta solución:

```
string &operator=(const string &cad)
```

Asigna la secuencia de caracteres en *cad* a la **string** que invoca. Devuelve una referencia al objeto que invoca. Otras versiones del operador de asignación le permiten asignar una cadena terminada en un carácter nulo o un carácter a un objeto **string**.

Puede unir una **string** con otra al usar el operador **+**. Se define como una función que no es miembro. Tiene varias formas. He aquí la usada en esta solución:

```
string operator+(const string &izqarr, const string &derarr)
```

Une *derarr* con *izqarr* y devuelve un objeto **string** que contiene el resultado. Otras versiones del operador de unión le permiten unir un objeto **string** con una cadena terminada en un carácter nulo o con un carácter.

Puede insertar una cadena en otra al usar la función **insert()**. Tiene varias formas. La usada aquí es:

```
string &insert(size_type ind, const string &cad)
```

Inserta *cad* en la cadena que invoca en el índice especificado por *ind*. Devuelve una referencia al objeto que invoca.

Todos los operadores relacionales están definidos en la clase **string** por funciones de operador que no son miembro. Realizan comparaciones lexicográficas de las secuencias de caracteres contenidas dentro de dos cadenas. Cada operador tiene varias formas sobrecargadas. Los operadores usados aquí son **==**, **<=** y **>**, pero todos los operadores relacionales funcionan de la misma manera básica. He aquí las versiones de estos operadores que se usan en esta solución:

```
bool operator==(const string &izqarr, const string &derarr)
bool operator<=(const string &izqarr, const string &derarr)
bool operator>(const string &izqarr, const string &derarr)
```

En todos los casos, *izqarr* se refiere al operando de la izquierda y *derarr* al de la derecha. Se devuelve el resultado de la comparación. Otras versiones de estos operadores le permiten comparar un objeto **string** con una cadena terminada en un carácter nulo.

Puede obtener una referencia a un elemento específico en una **string** al usar el operador de indización de la matriz []. Está implementado como una función miembro, como se muestra aquí:

char &operator[](size_type *ind*)

Devuelve una referencia al carácter en el índice basado en cero especificado por *ind*. Por ejemplo, dado un objeto **string** llamado **micad**, la expresión **micad[2]** devuelve una referencia al tercer carácter en **micad**. También está disponible una versión con **const**.

El número de caracteres contenido en la cadena puede obtenerse al llamar a **size()**, como se muestra aquí:

size_type size() const

Devuelve el número de caracteres en la cadena. Como se explicó en la revisión general, al principio de este capítulo, **size_type** es un elemento de **typedef** que representa alguna forma de entero sin signo.

El número de caracteres que un objeto **string** puede contener no está predeterminado. En cambio, el objeto crecerá de acuerdo con las necesidades para adecuarse al tamaño de la cadena que habrá de encapsular. Sin embargo, todos los objetos **string** empiezan con una capacidad inicial, que es el número máximo de caracteres que puede contener antes de que deba asignarse más memoria. La capacidad de un objeto **string** puede determinarse al llamar a **capacity()**, que se muestra aquí:

size_type capacity() const

Devuelve la capacidad actual de la **string** que invoca.

La capacidad de un objeto **string** puede ser importante porque las asignaciones de memoria ocupan mucho tiempo. Si sabe por anticipado el número de caracteres que contendrá una **string**, entonces puede establecer la capacidad en esa cantidad, eliminando así una reasignación de memoria. Para esto, llame a **reserve()**, que se muestra a continuación:

void reserve(size_type *num* = 0)

Establece la capacidad de la **string** que invoca para que sea por lo menos igual a *num*. Si *num* es menor o igual al número de caracteres en la cadena, entonces la llamada a **reserve()** es una solicitud para reducir la capacidad a un tamaño igual. Sin embargo, esta solicitud puede ignorarse.

Puede eliminar uno o más caracteres de una cadena al llamar a **erase()**. Hay tres versiones de **erase()**. Aquí se muestra la que se usa en esta solución:

string &erase(size_type *ind* = 0, size_type *long* = npos)

A partir de *ind*, elimina *long* caracteres del objeto que invoca. Devuelve una referencia al objeto que invoca.

Una de las funciones miembro de **string** más interesantes es **push_back()**. Agrega un carácter al final de la cadena:

void push_back (const char *car*)

Agrega *car* al final de la cadena que invoca. Es muy útil cuando quiere crear una cola de caracteres.

Puede obtener una parte de una cadena (es decir, una subcadena), al llamar a **substr()**, que se muestra aquí:

string substr(size_type *ind* = 0, size_type *long* = npos) const

Devuelve una subcadena de *long* caracteres, empezando en *ind* dentro de la **string** que invoca.

Ejemplo

En el siguiente ejemplo se ilustran varias de las operaciones fundamentales con cadenas:

```
// Demuestra las operaciones básicas con cadenas.
#include <iostream>
#include <string>

using namespace std;

int main()
{
  // Crea algunos objetos de cadena. Tres se inicializan
  // usando la literal de cadena pasada como argumento.
  string cad1("Alfa");
  string cad2("Beta");
  string cad3("Gamma");
  string cad4;

  // Salida de una cadena vía cout.
  cout << "Las cadenas originales son:\n";
  cout << "  cad1: " << cad1 << endl ;
  cout << "  cad2: " << cad2 << endl ;
  cout << "  cad3: " << cad3 << "\n\n";

  // Despliega la longitud máxima de la cadena.
  cout << "La longitud m\u00a0xima de la cadena es: " << cad1.max_size()
       << "\n\n";

  // Despliega el tamaño de cad1.
  cout << "cad1 contiene " << cad1.size() << " caracteres.\n";

  // Despliega la capacidad de cad1.
  cout << "Capacidad de cad1: " << cad1.capacity() << "\n\n";

  // Despliega los caracteres de una cadena, de uno en uno
  // empleando el operador de indización.
  for(unsigned i = 0; i < cad1.size(); ++i)
    cout << "cad1[i]: " << cad1[i] << endl;
  cout << endl;

  // Asigna una cadena a otra.
  cad4 = cad1;
  cout << "cad4 tras la asignaci\u00a2n de cad1: " << cad4 << "\n\n";
```

```
  // Une dos cadenas.
  cad4 = cad1 + cad3;
  cout << "cad4 tras la asignaci\u00a2n de cad1+cad3: " << cad4 << "\n\n";

  // Inserta una cadena en otra.
  cad4.insert(4, cad2);
  cout << "cad4 tras insertar cad2: " << cad4 << "\n\n";

  // Obtiene una subcadena.
  cad4 = cad4.substr(4, 4);
  cout << "cad4 tras la asignaci\u00a2n de cad4.substr(4, 3): "
       << cad4 << "\n\n";

  // Compara dos cadenas.
  cout << "Compara cadenas.\n";
  if(cad3 > cad1) cout << "cad3 > cad1\n";
  if(cad3 == cad1+cad2)
    cout << "cad3 == cad1+cad2\n";
  if(cad1 <= cad2)
    cout << "cad1 <= cad2\n\n";

  // Crea un objeto de cadena usando otro.
  cout << "Inicializa cad5 con el contenido de cad1.\n";
  string cad5(cad1);
  cout << "cad5: " << cad5 << "\n\n";

  // Borra cad4.
  cout << "Borrando cad4.\n";
  cad4.erase();
  if(cad4.empty()) cout << "cad4 est\u00a0 ahora vac\u00a1a.\n";
  cout << "El tama\u00a4o y la capacidad de cad4 son " << cad4.size() << " "
       << cad4.capacity() << "\n\n";

  // Usa push_back() para agregar caracteres a cad4.
  for(char ch = 'A'; ch <= 'Z'; ++ch)
    cad4.push_back(ch);
  cout << "cad4 tras llamar a push_back(): " << cad4 << endl;
  cout << "El tama\u00a4o y la capacidad de cad4 son ahora " << cad4.size() << " "
       << cad4.capacity() << "\n\n";

  // Establece la capacidad de cad4 en 128.
  cout << "Estableciendo la capacidad de cad4 en 128\n";
  cad4.reserve(128);
  cout << "La capacidad de cad4 es ahora: " << cad4.capacity() << "\n\n";

  // Ingresa una cadena vía cin.
  cout << "Ingrese una cadena: ";
  cin >> cad1;
  cout << "Introdujo: " << cad1 << "\n\n";

  return 0;
}
```

Aquí se muestra la salida:

```
Las cadenas originales son:
  cad1: Alfa
  cad2: Beta
  cad3: Gamma

La longitud máxima de la cadena es: 4294967294

cad1 contiene 4 caracteres.
Capacidad de cad1: 15

cad1[i]: A
cad1[i]: l
cad1[i]: f
cad1[i]: a

cad4 tras la asignación de cad1: Alfa

cad4 tras la asignación de cad1+cad3: AlfaGamma

cad4 tras insertar cad2: AlfaBetaGamma

cad4 tras la asignación de cad4.substr(4, 3): Beta

Compara cadenas.
cad3 > cad1
cad1 <= cad2

Inicializa cad5 con el contenido de cad1.
cad5: Alfa

Borrando cad4.
cad4 está ahora vacía.
El tamaño y la capacidad de cad4 son 0 15

cad4 tras llamar a push_back(): ABCDEFGHIJKLMNOPQRSTUVWXYZ
El tamaño y la capacidad de cad4 son ahora 26 31

Estableciendo la capacidad de cad4 en 128
La capacidad de cad4 es ahora: 143

Ingrese una cadena: prueba
Introdujo: prueba
```

Tal vez lo más importante que hay que notar en el ejemplo es que el tamaño de las cadenas no está especificado. Como se explicó, los objetos **string** reciben un tamaño automáticamente para contener la cadena que se les da. Por tanto, cuando se asignan o unen cadenas, la cadena de destino crecerá de acuerdo con lo necesario para acomodar el tamaño de la nueva cadena. No es posible rebasar el final de la cadena. Este aspecto dinámico de los objetos **string** es una de las maneras en que son mejores que las cadenas terminadas en un carácter nulo estándar, que *son* el tema de los rebases de límite. (Como se mencionó en la revisión general, un intento de crear **string** que excede la cadena más larga posible da como resultado que se lance **lenght_error**. Por tanto, no es posible rebasar **string()**.)

Hay otro elemento importante que debe tomarse en cuenta en la ejecución de ejemplo. Cuando la capacidad de **cad4** se aumenta, al llamar a **reserve()** con un argumento de 128, la capacidad real se vuelve 143. Recuerde que una llamada a **reserve()** causa que la capacidad aumente *por lo menos* al valor especificado. La implementación es libre de asignarle un valor más elevado. Esto podría suceder porque las asignaciones podían ser más eficientes en bloques de cierto tamaño, por ejemplo. (Por supuesto, debido a las diferencias entre compiladores, tal vez vea un valor de capacidad diferente cuando ejecute su programa de ejemplo. Son de esperar esas diferencias.)

Opciones

Incluso para las operaciones básicas con cadenas, **string** ofrece muchas opciones. Aquí se mencionan varias de ellas.

Como se explicó, para obtener el número de caracteres mantenido actualmente por una cadena, puede llamar a **size()**. Sin embargo, también puede llamar a **lenght()**. Devuelven el mismo valor y funcionan de la misma manera. En esencia, **size()** y **lenght()** son simplemente dos nombres diferentes para la misma función. La razón para los dos nombres es histórica. Todos los contenedores STL deben implementar el método **size()**. Aunque no siempre se considera como parte de STL, **string** cumple con todos los requisitos para un contenedor y es compatible con STL. Parte de estos requisitos es que un contenedor debe proporcionar una función **size()**. Por tanto, **size()** se vuelve parte de **string**.

La función **insert()** tiene varias formas adicionales. Por ejemplo, puede insertar una parte de una **string** en otra, uno o más caracteres en una **string**, o una cadena terminada en un carácter nulo en una **string**.

La función **erase()** tiene varias formas adicionales que le permiten eliminar caracteres a los que hace referencia un iterador (consulte *Opere en objetos **string** mediante iteradores*).

Aunque el uso del operador de indización [] es más sencillo, también puede obtener una referencia a un carácter específico al llamar al método **at()**. Aquí se muestra cómo se implementa para **string**:

char &at(size_type *ind*)

Devuelve una referencia al carácter en el índice basado en cero especificado por *ind*. También está disponible una versión de **const**.

Como se muestra en la solución, pueden realizarse asignaciones y uniones simples empleando los operadores **=** y **+** definidos por **string**. En casos en que se necesitan asignaciones o uniones más sofisticadas, **string** proporciona las funciones **assign()** y **append()**. Estas funciones tienen muchas formas que le permiten asignar o adjuntar partes de una cadena, toda una cadena terminada en un carácter nulo (o parte de ella), o uno o más caracteres. También hay formas que dan soporte a iteradores. Aunque éstos son demasiados para describirse en una solución, he aquí un ejemplo de cada una:

string &assign(const string &*cad*, size_type *ind*, size_type *long*)
string &append(const string &*cad*, size_type *ind*, size_type *long*)

La versión de **assign()** asigna una subcadena de *cad* a la cadena que invoca. La subcadena empieza en *ind* y se ejecuta por *long* caracteres. Esta versión de **append()** adjunta una subcadena de *cad* al final de la cadena que invoca. La subcadena empieza en *ind* y se ejecuta por *long* caracteres. Ambas funciones devuelven una referencia al objeto que invoca.

Los operadores relacionales son la manera más fácil de compartir una cadena con otra. Además de las formas usadas en la solución, otras versiones de estos operadores le permiten comparar un objeto **string** con una cadena terminada en un carácter nulo. Para proporcionar flexibilidad agregada, **string** también brinda la función **compare()**, que le permite comparar partes de dos cadenas. He aquí un ejemplo. Compara una cadena con una subcadena de la cadena que invoca.

int compare(size_type *ind*, size_type *long*, const string &*cad*) const

Esta función compara *cad* con la subcadena dentro de la cadena que invoca y que empieza en *ind* y tiene *long* caracteres de largo. Devuelve menos de cero si la secuencia en la cadena que invoca es menor que *cad*, cero si las dos secuencias son iguales, y mayor que cero si la secuencia en la cadena que invoca es mayor que *cad*.

Puede eliminar todos los caracteres de una **string** de dos maneras. En primer lugar, como se muestra en la solución, puede usar la función **erase()**, permitiendo los argumentos predeterminados. Como opción, puede llamar a **clear()**, que se muestra aquí:

void clear()

Busque un objeto string

Componentes clave		
Encabezado	**Clases**	**Funciones**
<string>	string	size_type find(const char *cad, size_type ind = 0) const size_type find(const string *cad, size_type ind = 0) const size_type find_first_of(const char *cad, size_type ind = 0) const size_type find_first_of(const string *cad, size_type ind = 0) const size_type find_first_not_of(const char *cad, size_type ind = 0) const size_type find_last_of(const char *cad, size_type ind = npos) const size_type find_last_not_of(const char *cad, size_type ind = npos) const size_type rfind(const char *cad, size_type ind = npos) const

La clase **string** define una serie poderosa de funciones que busca una cadena. Estas funciones le permiten encontrar:

- La primera aparición de una subcadena o un carácter.
- La última aparición de una subcadena o un carácter.
- La primera aparición de cualquier carácter en un conjunto de caracteres.
- La última aparición de cualquier carácter en un conjunto de caracteres.

- La primera aparición de cualquier carácter que no es parte de un conjunto de caracteres.
- La última aparición de cualquier carácter que no es parte de un conjunto de caracteres.

En esta solución se demuestra su uso.

Paso a paso

La búsqueda de una **string** incluye estos pasos:

1. Para encontrar la primera aparición de una secuencia o carácter, llame a **find()**.
2. Para encontrar la última aparición de una secuencia o carácter, llame a r**find()**.
3. Para encontrar la primera aparición de cualquier carácter en un conjunto de caracteres, llame a **find_first_of()**.
4. Para encontrar la última aparición de cualquier carácter en un conjunto de caracteres, llame a **find_last_of()**.
5. Para encontrar la primera aparición de cualquier carácter que no es parte de un conjunto de caracteres, llame a **find_first_not_of()**.
6. Para encontrar la última aparición de cualquier carácter que no es parte de un conjunto de caracteres, llame a **find_last_not_of()**.

Análisis

Todas las funciones de búsqueda tienen cuatro formas, lo que les permite especificar el objetivo de la búsqueda como una **string**, una cadena terminada en un carácter nulo, una porción de una cadena terminada en un carácter nulo, o un carácter. Aquí se describen las formas usadas por los ejemplos de esta solución.

La función **find()** encuentra la primera aparición de una subcadena o un carácter dentro de otra cadena. Aquí están las formas usadas en esta solución o en el Ejemplo adicional:

size_type find(const string &*cad*, size_type *ind* = 0) const
size_type find(const char **cad*, size_type *ind* = 0) const

Ambas devuelven el índice de la primera aparición de *cad* dentro de la cadena que invoca. El parámetro *ind* especifica el índice en que empezará la búsqueda dentro de la cadena que invoca. En la primera forma, *cad* es una referencia a una **string**. En la segunda forma, *cad* es un apuntador a una cadena terminada en un carácter nulo. Si no se encuentra una coincidencia, se devuelve **npos**.

La función **rfind()** encuentra la última aparición de una subcadena o un carácter dentro de otra cadena. La forma que se usa aquí es:

size_type rfind(const char **cad*, size_type *ind* = npos) const

Devuelve el índice de la última aparición de *cad* dentro de la cadena que invoca. El parámetro *ind* especifica el índice en que empezará la búsqueda dentro de la cadena que invoca. Si no se encuentra una coincidencia, se devuelve **npos**.

Para encontrar la primera aparición de cualquier carácter dentro de un conjunto de caracteres, se llama a **find_first_of()**. He aquí las formas usadas en esta solución o en el Ejemplo adicional:

size_type find_first_of(const string &*cad*, size_type *ind* = 0) const
size_type find_first_of(const char **cad*, size_type *ind* = 0) const

Ambos devuelven el índice del primer carácter dentro de la cadena que invoca y que coincide con cualquier carácter en *cad*. La búsqueda empieza en el índice *ind*. Se devuelve **npos** si no se encuentra una coincidencia. La diferencia entre las dos es simplemente el tipo de *cad*, que puede ser **string** o una cadena terminada en un carácter nulo.

Para encontrar la primera aparición de cualquier carácter que no es parte de un conjunto de caracteres, llame a **find_first_not_of()**. He aquí las formas usadas en esta solución o en el Ejemplo adicional:

size_type find_first_not_of(const string &*cad*, size_type *ind* = 0) const
size_type find_first_not_of(const char **cad*, size_type *ind* = 0) const

Ambas devuelven el índice del primer carácter dentro de la cadena que invoca y que *no* coincide con cualquier carácter en *cad*. La búsqueda empieza en el índice *ind*. Se devuelve **npops** si no se encuentra una coincidencia. La diferencia entre ambas es simplemente el tipo de *cad*, que puede ser **string** o una cadena terminada en un carácter nulo.

Para encontrar la última aparición de cualquier carácter que no es parte de un conjunto de caracteres, llame a **find_last_of()**. Ésta es la forma usada en la solución:

size_type find_last_of(const char **cad*, size_type *ind* = npos) const

Devuelve el índice del último carácter dentro de la cadena que invoca y que coincide con cualquier carácter en *cad*. La búsqueda empieza en el índice *ind*. Se devuelve **npos** si no se encuentra una coincidencia.

Para encontrar la última aparición de cualquier carácter que no es parte del conjunto de caracteres, llame a **find_last_not_of()**. La forma usada en esta solución es:

size_type find_last_not_of(const char **cad*, size_type *ind* = npos) const

Devuelve el índice del último carácter dentro de la cadena que invoca y que *no* coincide con cualquier carácter en *cad*. La búsqueda empieza en el índice *ind*. Se devuelve **npos** si no se encuentra una coincidencia.

NOTA *Como se acaba de describir, se devuelve **npos** con las funciones **find…** cuando no se encuentra una coincidencia. La variable **npos** es de tipo **string::size_type**, que es alguna forma de entero sin signo. Sin embargo, **npos** se inicializa en –1. Esto causa que **npos** contenga su valor sin signo más largo posible. Microsoft recomienda que si estará comparando el valor de alguna variable con **npos**, entonces esa variable debe declararse de tipo **string::size_type**, en lugar de **int** o **unsigned** para asegurar que la comparación se maneja correctamente en todos los casos. Éste es el método empleado en estas soluciones. Sin embargo, no es poco común ver código en que **npos** se declara como **int** o **unsigned**.*

Ejemplo

En el siguiente ejemplo se muestran las funciones de búsqueda en acción:

```
// Busca en una cadena.
#include <iostream>
#include <string>
```

```
using namespace std;
void mostrarresultado(string s, string::size_type i);

int main()
{
  string::size_type ind;

  // Crea una cadena.
  string cad("uno dos tres, uno dos tres");
  string cad2;

  cout << "La cadena en que se busca: " << cad << "\n\n";

  cout << "Buscando el primer caso de 'dos'\n";
  ind = cad.find("dos");
  mostrarresultado(cad, ind);

  cout << "Buscando el \u00a3ltimo cso de 'dos'\n";
  ind = cad.rfind("dos");
  mostrarresultado(cad, ind);

  cout << "Buscado el primer caso de t o r\n";
  ind = cad.find_first_of("tr");
  mostrarresultado(cad, ind);

  cout << "Buscando el \u00a3ltimo csso de t o r\n";
  ind = cad.find_last_of("tr");
  mostrarresultado(cad, ind);

  cout << "Buscando el primer caso de cualquier car\u00a0cter diferente "
       << "de u, n, o, o espacio\n";
  ind = cad.find_first_not_of("uno ");
  mostrarresultado(cad, ind);

  cout << "Buscando el \u00a3ltimo caso de cualquier car\u00a0cter diferente "
       << "de u, n, o, o espacio\n";
  ind = cad.find_last_not_of("uno ");
  mostrarresultado(cad, ind);

  return 0;
}

// Despliega el resultado de la búsqueda.
void mostrarresultado(string s, string::size_type i) {

  if(i == string::npos) {
    cout << "No se ha encontrado alguna coincidencia.\n";
    return;
  }

  cout << "Se encontr\u00a2 una coincidencia en el \u00a1ndice " << i << endl;

  cout << "Cadena restante desde el punto de coincidencia: "
       << s.substr(i) << "\n\n";
}
```

Aquí se muestra la salida:

```
La cadena en que se busca: uno dos tres, uno dos tres

Buscando el primer caso de 'dos'
Se encontró una coincidencia en el índice 4
Cadena restante desde el punto de coincidencia: dos tres, uno dos tres

Buscando el último caso de 'dos'
Se encontró una coincidencia en el índice 18
Cadena restante desde el punto de coincidencia: dos tres

Buscado el primer caso de t or r
Se encontró una coincidencia en el índice 8
Cadena restante desde el punto de coincidencia: tres, uno dos tres

Buscando el último caso de t o r
Se encontró una coincidencia en el índice 23
Cadena restante desde el punto de coincidencia: tres

Buscando el primer caso de cualquier carácter diferente de u, n, o, o espacio
Se encontró una coincidencia en el índice 4
Cadena restante desde el punto de coincidencia: dos tres, uno dos tres

Buscando el último caso de cualquier carácter diferente de u, n, o, o espacio
Se encontró una coincidencia en el índice 25
Cadena restante desde el punto de coincidencia: s
```

Ejemplo adicional: una clase de conversión en fichas para objetos string

La biblioteca estándar de C++ contiene la función **strtok()**, que puede usarse para convertir en fichas una cadena terminada en un carácter nulo (consulte *Convierta en fichas una cadena terminada en un carácter nulo*). Sin embargo, la clase **string** no define un equivalente correspondiente. Por fortuna, es muy fácil crear uno. Antes de empezar, es importante establecer que hay varias maneras diferentes de realizar esta tarea. En este ejemplo se muestra una de varias.

En el siguiente programa se crea una clase llamada **convertirenfichas** que encapsula esta función. Para convertir en fichas una cadena, construya primero una **convertirenfichas**, pasando la cadena como un argumento. A continuación, llame a **obtener_ficha()** para obtener las fichas individuales en la cadena. Los delimitadores que definen los límites de cada ficha se pasan a **obtener_ficha()** como cadena. Los delimitadores pueden cambiarse con cada llamada a **obtener_ficha()**. Esta función devuelve una cadena vacía cuando no hay más fichas para devolver. Observe que se usan las funciones **find_first_of()** y **find_first_not_of()**.

```
// Crea una clase llamada convertirenfichas que hace lo indicado.
#include <iostream>
#include <string>

using namespace std;

// La clase convertirenfichas se usa para la acción correspondiente.
// Pasa al constructor la cadena que habrá de convertirse en fichas.
// Para obtener la siguiente ficha, llame a obtener_ficha(),
// pasándolo en una cadena que llama al delimitador.
```

```
class convertirenfichas {
  string s;
  string::size_type indinicio;
  string::size_type indfinal;

public:
  convertirenfichas(const string &cad) {
    s = cad;
    indinicio = 0;
  }

  // Devuelve una ficha desde la cadena.
  string obtener_ficha(const string &delims);
};

// Devuelve una ficha desde la cadena. Devuelve
// una cadena vacía cuando ya no se encuentran más fichas.
// Pasa los delimitadores en delims.
string convertirenfichas::obtener_ficha(const string &delims) {

  // Devuelve una cadena vacía cuando no hay más
  // fichas por regresar.
  if(indinicio == string::npos) return string("");

  // Empezando en indinicio, encuentra el siguiente delimitador.
  indfinal = s.find_first_of(delims, indinicio);

  // Construye una cadena que contiene la ficha.
  string fic(s.substr(indinicio, indfinal-indinicio));

  // Encuentra el inicio de la siguiente ficha. Es un
  // carácter que no es un delimitador.
  indinicio = s.find_first_not_of(delims, indfinal);

  // Devuelve la siguiente ficha.
  return fic;
}

int main()
{
  // Cadenas que habrán de convertirse en fichas.
  string cadA("Yo tengo cuatro, cinco, seis fichas. ");
  string cadB("Tal vez tenga m\u00a0s fichas.\n\u00a8Y t\u00a3?");

  // Estas cadenas contienen los delimitadores.
  string delimitadores(" ,.\u00a8?\n");

  // Esta cadena contendrá la siguiente ficha.
  string ficha;

  // Crea dos convertirenfichas.
  convertirenfichas ficA(cadA);
  convertirenfichas ficB(cadB);
  // Despliega las fichas en cadA.
```

```
  cout << "Las fichas en cadA:\n";
  ficha = ficA.obtener_ficha(delimitadores);
  while(ficha != "") {
    cout << ficha << endl;
    ficha = ficA.obtener_ficha(delimitadores);
  }
  cout << endl;

  // Despliega las fichas en cadB.
  cout << "Las fichas en cadB:\n";
  ficha = ficB.obtener_ficha(delimitadores);
  while(ficha != "") {
    cout << ficha << endl;
    ficha = ficB.obtener_ficha(delimitadores);
  }

  return 0;
}
```

He aquí la salida:

```
Las fichas en cadA:
Yo
tengo
cuatro
cinco
seis
fichas

Las fichas en cadB:
Tal
vez
tenga
más
fichas
Y
tú
```

Hay una fácil mejora que tal vez quiera hacer a **convertirenfichas**: una función **reset()**. Esta función podría llamarse para habilitar una cadena que vuelva a convertirse en fichas desde el inicio. Esto es fácil. Simplemente establezca **indinicio** en cero, como se muestra aquí:

```
void rest() { indinicio = 0; }
```

Opciones

Como se mencionó, cada una de las funciones de **find…** tienen cuatro formas. Por ejemplo, he aquí todas las formas de **find()**:

```
size_type find(const string &cad, size_type ind = 0) const
size_type find(const char *cad, size_type ind = 0) const
size_type find(const char *cad, size_type ind, size_type long) const
size_type find(char cad, size_type ind = 0) const
```

Las primeras dos formas se describieron antes. La tercera busca la primera aparición de los primeros *long* caracteres. La cuarta busca la primera de *car*. En todos los casos, la búsqueda empieza en el índice especificado por *ind* dentro de la **string** que invoca, y se devuelve el índice en que se encuentra la coincidencia. Si no se encuentra una, se devuelve **npos**. Las otras funciones de **find...** tienen formas similares.

Como se mencionó en la revisión general, al principio del capítulo, la clase **string** satisface los requisitos generales para que sea un contenedor compatible con STL. Esto significa que los algoritmos declarados en **<algorithm>** pueden operar en él. Por tanto, puede hacerse una búsqueda en un objeto **string** al usar los algoritmos de búsqueda, como **search()**, **find()**, **find_first_of()**, etc. La ventaja que ofrecen los algoritmos es la capacidad de proporcionar un predicado definido por el usuario que le permite especificar cuando un carácter en la cadena coincide con otro. Esta característica se utiliza en la solución *Cree una búsqueda intensiva de mayúsculas y minúsculas y funciones de búsqueda y reemplazo para objetos **string*** para implementar una función de búsqueda que ignora las diferencias entre mayúsculas y minúsculas. (STL y los algoritmos se cubren a fondo en los capítulos 3 y 4.)

Cree una función de búsqueda y reemplazo para objetos string

Componentes clave		
Encabezado	**Clases**	**Funciones**
<string>	string	size_type find(const string &*cad*, size_type *ind* = 0) const string &replace(size_type *ind*, size_type *long*, const string &*cad*)

La clase **string** proporciona soporte muy rico para el reemplazo de una subcadena con otra. Esta operación es proporcionada por la función **replace()**, de la que hay diez formas. Éstas le dan una gran flexibilidad en la especificación de la manera en que el proceso de reemplazo tendrá lugar. Por ejemplo, puede especificar la cadena de reemplazo como un objeto **string** o una cadena terminada en un carácter nulo. Puede especificar cuál parte de la cadena que invoca se reemplazará al especificar índices o mediante el uso de iteradores. En ésta solución se usa **replace()** junto con la función **find()**, que se demostró en la anterior, para implementar una función de búsqueda y reemplazo para objetos **string**. Como verá, debido al soporte que **string** proporciona mediante **find()** y **replace()**, la implementación de la búsqueda y reemplazo es simple. También representa una implementación mucho más clara que la misma función implementada para cadenas terminadas en un carácter nulo. (Consulte *Cree una función de búsqueda y reemplazo para cadenas terminadas en un carácter nulo*.)

Paso a paso

Crear una función de búsqueda y reemplazo para objetos **string** implica estos pasos:

1. Cree una función llamada **buscar_y_reemplazar()** que tenga este prototipo:

```
bool buscar_y_reemplazar(string &cad, const string &subcadant,
                         const string $subcadnue);
```

La cadena que habrá de cambiarse se pasa vía **cad**. La subcadena que se reemplazará se pasa en **subcadant**. El reemplazo se pasa en **subcadnue**.
2. Use la función **find()** para encontrar la primera aparición de **subcadant**.
3. Use la función **replace()** para sustituir **subcadnue**.
4. Devuelve el valor true si se realizó un reemplazo y false, si no.

Análisis

El método **find()** se describió en la solución anterior y el análisis no se repetirá aquí.

Una vez que se ha encontrado la subcadena, puede reemplazarse al llamar a **replace()**. Hay diez formas de **replace()**. Aquí se muestra la usada en esta solución:

string &replace(size_type *ind*, size_type *long*, const string &*cad*)

Empezando en *ind*, dentro de la cadena que invoca, esta versión reemplaza hasta *long* caracteres con la cadena en *cad*. La razón de que reemplace "hasta" *long* caracteres es que no es posible reemplazar más allá del final de la cadena. Por tanto, si *long* + *ind* excede la longitud total de la cadena, sólo se reemplazarán los caracteres de *ind* al final. La función devuelve una referencia a la cadena que invoca.

Ejemplo

He aquí la manera de implementar la función **buscar_y_reemplazar()**:

```
// En la cadena a la que hace referencia cad, reemplaza subcadant con subcadnue.
// Por tanto, esta función modifica la cadena a la que hace referencia cad.
// Devuelve true si ocurre un reemplazo, y false si no.
bool buscar_y_reemplazar(string &cad, const string &subcadant,
                         const string &subcadnue) {
  string::size_type indinicio;

  indinicio = cad.find(subcadant);

  if(indinicio != string::npos) {
    cad.replace(indinicio, subcadant.size(), subcadnue);
    return true;
  }

  return false;
}
```

Si compara esta versión de **buscar_y_reemplazar()** con la creada para cadenas terminadas en un carácter nulo, verá que ésta es mucho más pequeña y simple. Hay dos razones para esto. En primer lugar, porque los objetos de tipo **string** son dinámicos: pueden crecer o reducirse de acuerdo con las necesidades.

Por tanto, es fácil reemplazar una subcadena con otra. No es necesario preocuparse de rebasar el límite de la matriz cuando la longitud de la cadena aumenta, por ejemplo. En segundo lugar, **string** proporciona una función **replace()** que maneja automáticamente la eliminación de la subcadena antigua y la inserción de la nueva. Esto no se necesita para manejarse manualmente, como en el caso de la inserción de una cadena terminada en un carácter nulo.

En el siguiente ejemplo se muestra la función **buscar_y_reemplazar()** en acción:

```
// Implementa la opción de búsqueda y reemplazo para objetos de cadena.
#include <iostream>
#include <string>

using namespace std;

bool buscar_y_reemplazar(string &cad, const string &subcadant,
                        const string &subcadnue);

int main()
{

  string cad = "Si esto es una prueba, s\u00a2lo esto es.";

  cout << "Cadena original: " << cad << "\n\n";

  cout << "Reemplazando 'es'con 'fue':\n";

  // Lo siguiente reemplaza es con fue. Tome nota de que
  // pasa literales de cadena para las subcadenas.
  // Se convierten automáticamente en objetos de cadena.
  while(buscar_y_reemplazar(cad, "es", "fue"))
   cout << cad << endl;

  cout << endl;

  // Por supuesto, también puede pasar explícitamente objetos de cadena.
  string cadant("s\u00a2lo");
  string cadnue("entonces s\u00a2lo");
  cout << "Reemplaza 's\u00a2lo' con 'entonces s\u00a2lo'" << endl;
  buscar_y_reemplazar(cad, cadant, cadnue);
  cout << cad << endl;

  return 0;
}

// En la cadena a la que hace referencia cad, reemplaza subcadant con subcadnue.
// Por tanto, esta función modifica la cadena a la que hace referencia cad.
// Devuelve true si ocurre un reemplazo, y false si no.
bool buscar_y_reemplazar(string &cad, const string &subcadant,
                        const string &subcadnue) {
  string::size_type indinicio;

  indinicio = cad.find(subcadant);
```

```
  if(indinicio != string::npos) {
    cad.replace(indinicio, subcadant.size(), subcadnue);
    return true;
  }

  return false;
}
```

Aquí se muestra la salida:

```
Cadena original: Si esto es una prueba, sólo esto es.

Reemplazando 'es' con 'fue':
Si fueto es una prueba, sólo esto es.
Si fueto fue una prueba, sólo esto es.
Si fueto fue una prueba, sólo fueto es.
Si fueto fue una prueba, sólo fueto fue.

Reemplaza 'sólo' con 'entonces sólo'
Si fueto fue una prueba, entonces sólo fueto fue.
```

Opciones

La función **replace()** tiene otras formas diversas. Las tres que se usan con más frecuencia se describen aquí. Todas devuelven una referencia a la cadena que invoca.

La siguiente forma de **replace()** toma una cadena terminada en un carácter nulo como cadena de reemplazo:

string &replace(size_type *ind*, size_type *long*, const char **cad*)

Empezando en *ind* dentro de la cadena que invoca, reemplaza hasta *long* caracteres con la cadena en *cad*.

Para reemplazar una subcadena con una parte de otra cdm utilice esta forma:

string &replace(size_type *ind1*, size_type *long1*, const char **cad*
size_type *ind2*, size_type *long2*)

Reemplaza hasta *long1* caracteres en la cadena que invoca, empezando en *ind1*, con los *long2* caracteres a partir de la cadena en *cad*, empezando en *ind2*.

La siguiente forma de **replace()** opera en iteradores:

string &replace(iterator *inicio*, iterator *final*, const string &*cad*)

El rango especificado por *inicio* y *final* es reemplazado con los caracteres en *cad*.

La función **buscar_y_reemplazar()** opera de una manera sensible a mayúsculas y minúsculas. Es posible realizar una búsqueda y reemplazo sensible a mayúsculas y minúsculas, pero se requiere un poco de trabajo. Una manera es implementar una función de este tipo que use el algoritmo de STL **search()** estándar. Éste le permite especificar un predicado binario que puede hacerse a la medida para probar si dos caracteres son iguales con independencia de las diferencias entre mayúsculas y minúsculas. Luego puede usar esta función para encontrar la ubicación de la subcadena que habrá de eliminarse. Para ver este método en acción, consulte *Cree una búsqueda no sensible a mayúsculas y minúsculas y funciones de búsqueda y reemplazo para objetos **string***.

Opere en objetos string mediante iteradores

Componentes clave		
Encabezado	**Clases**	**Funciones**
<string>	string	iterator begin() iterator end() reverse_iterator rbegin() reverse_iterator rend() iterator erase(iterator *inicio*, iterator *final*) template <class InIter> void insert(iterator *itr*, InIter *inicio*, InIter *final*) string &replace(iterator *inicio*, iterator *final*, const char **cad*)
<algorithm>		template <class InIter, class T> InIter find(InIter *inicio*, InIter *final*, const T &*val*) template <class InIter, class OutIter, class Func> OutIter transform(InIter *inicio*, InIter *final*, OutIter *resultado*, Func *funUnaria*)

En esta solución se muestra cómo usar los iteradores con objetos de tipo **string**. Como la mayoría de los lectores sabe, los iteradores son objetos que actúan como apuntadores. Le dan la capacidad de hacer referencia a los contenidos del contenedor al usar una sintaxis parecida a un apuntador. También son el mecanismo que deja que diferentes tipos de contenedores sean manejados de la misma manera y que permite diferentes tipos de contenedores para intercambiar datos. Son uno de los conceptos más poderosos de C++.

Como se explicó en la revisión general de **string**, cerca del principio de este capítulo, **basic_string** satisface los requisitos básicos de un contenedor. Por tanto, la especialización de **string** de **basic_string** es, en esencia, un contenedor de caracteres. Uno de los requisitos de todos los contenedores es que den soporte a iteradores. Al dar soporte a iteradores, **string** ofrece tres beneficios importantes:

1. Los iteradores pueden mejorar algunos tipos de operaciones de **string**.
2. Los iteradores permiten que los diveros algoritmos STL operen en objetos **string**.
3. Los iteradores permiten que **string** sea compatible con otros contenedores STL. Por ejemplo, mediante iteradores, puede copiar los caracteres de una **string** en un **vector** o construir una **string** a partir de caracteres almacenados en **deque**.

La clase **string** da soporte a todas las operaciones básicas de iterador. También proporciona versiones de varias de las funciones, como **insert()** y **replace()**, que están diseñadas para trabajar con iteradores. En esta solución se demuestran las operaciones básicas y tres funciones habilitadas

por iteradores, y se muestra la manera en que los iteradores permiten que **string** se integre en el marco conceptual general del STL.

NOTA *Para conocer un análisis detallado de los iteradores, consulte el capítulo 3, que presenta soluciones basadas en STL.*

Paso a paso

Para operar en una cadena mediante iteradores se requieren estos pasos:

1. Declare una variable que contendrá un iterador. Para ello, debe usar uno de los tipos de iterador definidos por **string**, como **iterator** o **reverse_iterator**.
2. Para obtener un iterador al principio de una cadena, llame a **begin()**.
3. Para obtener un iterador al final de una cadena, llame a **end()**.
4. Para obtener un iterador inverso al principio de la cadena invertida, llame a **rbegin()**.
5. Para obtener un iterador inverso al final de la cadena invertida, llame a **rend()**.
6. Puede recorrer en ciclo los caracteres de una cadena mediante un iterador de una manera muy parecida a la que puede usar para que un apuntador recorra en ciclo los elementos de una matriz.
7. Puede crear un objeto **string** que se inicializa con los caracteres señalados por un rango de iteradores. Entre otros usos, esto le permite construir una **string** que contiene elementos de otro tipo de contenedor, como **vector**.
8. Muchas de las funciones de **string** definen versiones que operan a través de iteradores. Las demostradas en esta solución son **erase()**, **insert()** y **replace()**. Le permiten eliminar, insertar y reemplazar caracteres dentro de los iteradores determinados de una cadena a los extremos de los caracteres.
9. Debido a que los algoritmos STL funcionan mediante iteradores, puede usar cualquiera de los algoritmos en objetos de tipo **string**. Aquí se demuestran dos: **find()** y **transform()**. Requieren el encabezado **<algorithm>**.

Análisis

Una revisión general de los iteradores se presenta en el capítulo 3, y esa información no se repite aquí. Sin embargo, es útil revisar unos cuantos puntos clave. En primer lugar, el objeto al que señala un iterador se accede mediante el operador * de la misma manera en que éste se usa para acceder al objeto señalado por un apuntador. Como se aplica a **string**, el objeto señalado por un iterador es un valor **char**. En segundo lugar, cuando se incrementa un iterador, señala al siguiente objeto del contenedor. Cuando se reduce, señala al objeto anterior. Para **string**, esto significa que el iterador señala al siguiente carácter o al anterior.

Hay dos estilos básicos de iteradores a los que da soporte **string**: directos o inversos. Cuando se incrementa, un iterador directo se acerca al final de la cadena y cuando se reduce, lo hace hacia el principio. Un iterador inverso trabaja al revés. Cuando un iterador inverso se incrementa, se mueve hacia el principio de la cadena y cuando se reduce se mueve hacia el final. De estos dos iteradores básicos, la clase **string** declara cuatro tipos básicos de iteradores que tienen los siguientes nombres de tipo:

iterator	Iterador que se mueve hacia adelante que puede leer y escribir a lo que señala.
const_iterator	Iterador que se mueve hacia adelante y que es de sólo lectura.
reverse_iterator	Iterador que se mueve hacia atrás que puede leer y escribir a lo que señala.
const_reverse_iterator	Iterador que se mueve hacia atrás y que es de sólo lectura.

En la solución sólo se usan **iterator** y **reverse_iterator**, pero los otros dos funcionan de la misma manera, excepto que no se puede escribir el objeto al que señalan.

En los análisis siguientes, las mismas funciones usan los nombres de tipo genérico **InIter** y **OutIter**. En este libro, **InIter** es un tipo de iterador que es, por lo menos, capaz de leer operaciones. **OutIter** es un tipo de iterador que es, por lo menos, capaz de escribir operaciones. (Otros tipos de iteradores se analizan en el capítulo 3.)

Para declarar un iterador para una **string**, use uno de los tipos antes mencionados. Por ejemplo:

```
string::iterator itr;
```

declara un iterador directo que no es **const** y que puede usarse con un objeto **string**.

Para obtener un iterador al principio de una cadena (que es el primer carácter de la cadena), llame a **begin()**. Para obtener un iterador que señale a *uno después* del final de la cadena, llame a **end()**. Por tanto, el último carácter de la cadena está en **end()–1.** Aquí se muestran estas funciones:

iterator begin()
iterator end()

La ventaja de que **end()** devuelva un iterador a uno después del último carácter es que pueden escribirse bucles muy eficientes que recorren en ciclo todos los caracteres de una cadena. He aquí un ejemplo:

```
string::iterator itr;
for(itr = cad.begin(); itr != cad.end(); ++itr) {
  // . . .
}
```

Cuando **itr** sea igual a **end()**, todos los caracteres de **cad** se habrán examinado.

Cuando se usa un iterador inverso, puede obtener uno al último carácter en la cadena al llamar a **rbegin()**. Para obtener un iterador inverso a uno antes del primer carácter en la cadena, llame a **rend()**. Se muestran aquí:

reverse_iterator rbegin()
reverse_iterator rend()

Se usa un iterador inverso de la misma manera en que usa un iterador regular. La única diferencia es que recorre la cadena en la dirección inversa.

La clase **string** proporciona un constructor que le permite crear una cadena que se inicializa con caracteres señalados por iteradores. Aquí se muestra:

template <class InIter> string(InIter *inicio*, InIter *final*,
const Allocator &*asig* = Allocator())

El rango de caracteres está especificado por *inicio* y *final*. El tipo de estos iteradores está especificado por el tipo genérico **InIter**, que indica que los iteradores deben dar soporte a operaciones de lectura. Sin embargo, no tienen que ser de tipo **string::iterator**. Esto significa que puede usar este constructor para crear una cadena que contenga caracteres de otro contenedor, como **vector**.

Varias de las funciones de **string** tienen formas sobrecargadas que utilizan iteradores para acceder al contenido de la cadena. En esta solución se usan tres que son representativas: **insert()**, **erase()** y **replace()**. A continuación se muestran las versiones usadas en esta solución:

iterator erase(iterator *inicio*, iterator *final*)
string &replace(iterator *inicio*, iterator *final*, const char **cad*)
template <class InIter>
  void insert(iterator *itr*, InIter *inicio*, InIter *final*)

El método **erase()** elimina los caracteres en el rango señalado por *inicio* a *final*. Devuelve un iterador al carácter que sigue al último carácter eliminado. La función **replace()** reemplaza con *cad* los caracteres en el rango especificado por *inicio* y *final*. Devuelve una referencia al objeto que invoca. (Otras versiones habilitadas por iterador de **replace()** le permiten pasar una **string** a *cad*.) El método **insert()** inserta los caracteres en el rango señalado por *inicio* y *final* inmediatamente antes del elemento especificado por *itr*. En **insert()**, observe que *inicio* y *final* son del tipo genérico **InIter**, lo que significa que los iteradores deben dar soporte a operaciones de lectura. Todos los tipos de iterador de **string** satisfacen esta restricción. Así lo hacen muchos otros iteradores. Por tanto, puede insertar caracteres de otro tipo de contenedor en una **string**. Ésta es una de las ventajas de los iteradores.

Debido a que los algoritmos STL funcionan mediante iteradores, puede usarlos en **string**. Los algoritmos STL están declarados en **<algorithm>** y realizan varias operaciones en contenedores. En esta solución se demuestra el uso de dos algoritmos, **find()** y **transform()**, que se muestran aquí:

template <class InIter, class T>
  InIter find(InIter *inicio*, InIter *final*, const T &*val*)

template <class InIter, class OutIter, class Func>
  OutIter transform(InIter *inicio*, InIter *final*, OutIter *resultado*, Func *funUnaria*)

El algoritmo **find()** busca el valor especificado por *val* en el rango señalado por *inicio* y *final*. Devuelve un iterador a la primera aparición del elemento o a *final*, si el valor no está en la secuencia. El algoritmo **transform()** aplica una función a un rango de elementos especificado por *inicio* y *final*, poniendo el resultado en *resultado*. La función que habrá de aplicarse está especificada en *funUnaria*. Esta función recibe un valor de la secuencia y debe regresar su transformación. Por tanto, los tipos de parámetro y de devolución deben ser compatibles con el tipo de objetos almacenados en el contenedor, que en el caso de **string** es **char**. El algoritmo **transform()** devuelve un iterador al final de la secuencia resultante. Observe que el resultado es de tipo **OutIter**, lo que significa que debe dar soporte a operaciones de escritura.

Ejemplo

En el siguiente ejemplo se muestra cómo usar iteradores con objetos **string**. También se demuestran versiones de iterador de las funciones miembro de **string insert()**, **replace()** y **find()**. Además se usan los algoritmos STL **find()** y **transform()**.

```
// Demuestra iteradores con cadenas.
#include <iostream>
#include <string>
#include <cctype>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
  string cadA("La prueba es la siguiente.");

  // Crea un iterador a una cadena.
  string::iterator itr;

  // Usa un iterador para recorrer en ciclo los caracteres
  // de una cadena.
  cout << "Despliega una cadena mediante un iterador.\n";
  for(itr = cadA.begin(); itr != cadA.end(); ++itr)
    cout << *itr;
  cout << "\n\n";

  // Usa un iterador inverso para desplegar la cadena invertida.
  cout << "Despliega una cadena invertida usando un iterador inverso.\n";
  string::reverse_iterator ritr;
  for(ritr = cadA.rbegin(); ritr != cadA.rend(); ++ritr)
    cout << *ritr;
  cout << "\n\n";

  // Inserta una cadena mediante un iterador.

  // Primero, usa el algoritmo STL find() para obtener un
  // iterador al principio de la primera 'a'.
  itr = find(cadA.begin(), cadA.end(), 'a');

  // Luego, incrementa el iterador para que señale al
  // carácter después de 'a', que en este caso es un espacio.
  ++itr;

  // Inserta en cad usando la versión de iterador de insert().
  cout <<"Inserta en una cadena mediante un iterador.\n";
  string cadB(" mayor");
  cadA.insert(itr, cadB.begin(), cadB.end());
  cout << cadA << "\n\n";

  // Ahora, reemplaza 'mayor' con 'mejor'.
  cout << "Reemplaza mayor con mejor.\n";
  itr = find(cadA.begin(), cadA.end(), 'm');
  cadA.replace(itr, itr+5, "mejor");
  cout << cadA << "\n\n";

  // Ahora, elimina ' mejor'.
  cout << "Elimina ' mejor'.\n";
```

```
  itr = find(cadA.begin(), cadA.end(), 'm');
  cadA.erase(itr, itr+6);
  cout << cadA << "\n\n";

  // Usa un iterador con el algoritmo STL transform() para convertir
  // una cadena a mayúsculas.
  cout << "Use el algoritmo STL transform() para convertir una "
       << "cadena en may\u00a3sculas.\n";
  transform(cadA.begin(), cadA.end(), cadA.begin(), toupper);
  cout << cadA << "\n\n";

  // Crea una cadena desde un vector<char>.
  vector<char> vec;
  for(int i=0; i < 10; ++i)
    vec.push_back('A'+i);

  string cadC(vec.begin(), vec.end());
  cout << "Se muestra cadC, construida a partir de un vector:\n";
  cout << cadC << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Despliega una cadena mediante un iterador.
La prueba es la siguiente.

Despliega una cadena invertida usando un iterador inverso.
.etneiugis al se abeurp aL

Inserta en una cadena mediante un iterador.
La mayor prueba es la siguiente.

Reemplaza mayor con mejor.
La mejor prueba es la siguiente.

Elimina ' mejor'.
La prueba es la siguiente.

Use el algoritmo STL transform() para convertir una cadena en mayúsculas.
LA PRUEBA ES LA SIGUIENTE.

Se muestra cadC, construida a partir de un vector:
ABCDEFGHIJ
```

Opciones

Como se mencionó, varias de las funciones miembro definidas por **string** tienen formas que operan en iteradores o que los devuelven. Además de **insert()**, **erase()** y **replace()** usadas en esta solución, **string** proporciona versiones habilitadas por iteradores de las funciones **append()** y **assign()**. Se muestran aquí:

template<class InIter> string &append(InIter *inicio*, InIter *final*)
template<class InIter> string &assign(InIter *inicio*, InIter *final*)

Esta versión de **append()** agrega la secuencia especificada por *inicio* y *final* al final de la cadena que invoca. Esta versión de **assign()** asigna la secuencia especificada por *inicio* y *final* a la cadena que invoca. Ambas devuelven una red a la cadena que invoca.

Cree una búsqueda no sensible a mayúsculas y minúsculas y funciones de búsqueda y reemplazo para objetos string

Componentes clave		
Encabezado	**Clases**	**Funciones**
<cctype>		int tolower(int *car*)
<string>	string	iterator begin() iterator end() string &replace(iterator *inicio*, iterator *final*, const string &*subcadnue*)
<algorithm>		template <class ForIter1, class ForIter2, class BinPred> ForIter3 search(ForIter1 *inicio1*, ForIter1 *final1*, ForIter2 *inicio2*, ForIter2 *final2*, BinPred *pfn*)

Aunque **string** es muy poderosa, no da soporte directo a dos funciones muy útiles. La primera es una función de búsqueda que ignora diferencias entre mayúsculas y minúsculas. Como casi todos los lectores saben, este tipo de búsqueda es una característica común y valiosa en muchos contextos. Por ejemplo, cuando se buscan coincidencias de la palabra "esto" en un documento, por lo general también querrá que se encuentre "Esto". La segunda es una función de búsqueda y reemplazo no sensible a mayúsculas y minúsculas, que reemplaza un subcadena con otra, independientemente de las diferencias entre mayúsculas y minúsculas. Puede usar este tipo de función, por ejemplo, para reemplazar instancias de "www" o "WWW" con las palabras "World Wide Web" en un solo paso. Cualquiera que sea el propósito, es fácil crear funciones de búsqueda y reemplazo que no sean sensibles a mayúsculas y minúsculas y que operan en objetos **string**. En esta solución se muestra una manera.

Las funciones desarrolladas por ésta, dependen de iteradores para acceder a los caracteres de una cadena. Debido a que **string** es un contenedor compatible con STL, proporciona soporte para iteradores. Este soporte es muy importante porque permite que una **string** se opere con algoritmos STL. Esta capacidad expande de manera significativa las maneras en que pueden modificarse las cadenas. También le permite crear soluciones mejoradas a las que, de otra manera, serían tareas más desafiantes. (Consulte la solución anterior para conocer información sobre el uso de iteradores con **string**.)

Paso a paso

Una manera de crear una función de búsqueda que ignora diferencias entre mayúsculas y minúsculas incluye estos pasos:

1. Cree una función de comparación llamada **comp_ign_mayus()** que realice una comparación no sensible a mayúsculas y minúsculas de dos valores **char**. He aquí un prototipo:

   ```
   bool comp_ign_mayus(char x, char y);
   ```

   Haga que la función devuelva un valor true si dos caracteres son iguales y false, si no.
2. Cree una función llamada **buscar_ign_mayus()** que tenga este prototipo:

   ```
   string::iterator buscar_ign_mayus(strinf &cad, const string &subcad);
   ```

   La cadena en que se buscará se pasa en **cad**. La subcadena que se buscará es **subcad**.
3. Dentro de **buscar_ign_mayus()**, use el algoritmo STL **search()** para buscar una subcadena en una cadena. Este algoritmo busca una coincidencia u otra en una secuencia. Las secuencias están especificadas por rangos de iteradores. Especifique la función **comp_ign_mayus()** creada en el paso 1 como el predicado binario que determina cuando un carácter es igual a otro. Esto permite que **search()** ignore diferencias entre mayúsculas y minúsculas cuando se busca. Observe que **search()** está declarada en el encabezado **<algorithm>**, que debe incluirse.
4. Haga que **buscar_ign_mayus()** devuelva un iterador al inicio de la primera coincidencia o **cad.end()**, si no se encuentra una coincidencia.

Para crear una función de búsqueda y reemplazo que ignore las diferencias entre mayúsculas y minúsculas, siga estos pasos:

1. Necesitará la función **buscar_ign_mayus()** descrita por los pasos anteriores. Por tanto, si aún no la ha creado, debe hacerlo en este momento.
2. Cree una función llamada **buscar_y_reemplazar_ign_mayus()** que tenga este prototipo:

   ```
   bool buscar_y_reemplazar_ign_mayus(string &cad, const string &subcadant,
                                      const string &subcadnue);
   ```

   La cadena que habrá de modificarse se pasa en **cad**. La cadena que habrá de reemplazarse se pasa en **subcadant**. La cadena con que se sustituirá se pasa en **subcadnue**.
3. Use **buscar_ign_mayus()** para encontrar la primera aparición de **subcadant** dentro de **cad**.
4. Use la versión de iterador de la función **replace()** de **string** para reemplazar la primera aparición de **subcadant** con **subcadnue**.
5. Haga que **buscar_y_reemplazar_ign_mayus()** devuelva el valor true si se hace el reemplazo y false si **cad** no contiene un caso de **subcadant**.

Análisis

Antes de que use el algoritmo **search()** para realizar una búsqueda no sensible a mayúsculas y minúsculas, debe crear una función que compara dos valores **char** de una manera independiente de mayúsculas y minúsculas. Debe regresar true si los caracteres son iguales y false si no lo son. En el lenguaje de STL, a esta función se le denomina *predicado binario*. (Consulte el capítulo 3 para conocer un análisis de los predicados binarios.)

Esta función se utiliza con el algoritmo **search()** para comparar dos elementos. Al hacer que ignore las diferencias entre mayúsculas y minúsculas, la búsqueda se realizará independientemente de estas diferencias. He aquí una manera de codificar esta función:

```
bool comp_ign_mayus(char x, char y) {
  return tolower(x) == tolower(y);
}
```

Observe que se utiliza la función **tolower()** para obtener el equivalente en minúsculas de cada carácter. (Consulte *Ignore diferencias entre mayúsculas y minúsculas cuando compare cadenas terminadas en un carácter nulo* para conocer detalles sobre **tolower()**.) Al convertir cada argumento a minúsculas, se eliminan las diferencias entre mayúsculas y minúsculas.

Para encontrar una subcadena, llame al algoritmo **search()**. Aquí se muestra la versión utilizada en esta solución:

template <class ForIter1, class ForIter2, class BinPred>
ForIter3 search(ForIter1 ***inicio1***, ForIter1 ***final1***,
ForIter2 ***inicio2***, ForIter2 ***final2***,
BinPred ***pfn***)

Busca una aparición de la secuencia especificada por *inicio2* y *final2* dentro del rango de la secuencia especificada por *inicio1* y *final1*. En este libro, los nombres de tipo genérico **ForIter1** y **ForIter2** indican iteradores que tienen capacidades de lectura/escritura y que puedan moverse hacia adelante. El predicado binario *pfn* determina cuando dos elementos son iguales. (En este libro, el nombre de tipo genérico **BinPred** indica un predicado binario.) Para los objetivos de la solución, pase **comp_ign_mayus()** a este parámetro. Si se encuentra una coincidencia, la función devuelve un iterador al principio de la secuencia coincidente. De otra manera, se devuelve *final1*.

La función **buscar_y_reemplazar_ign_mayus()** usa el iterador devuelto por **buscar_ign_mayus()** para encontrar la ubicación en que se sustituye una subcadena con otra. Para manejar el reemplazo real, puede usar esta versión de la función **replace()** de **string**, que opera mediante iteradores:

string &replace(iterator *inicio*, iterator *final*, const string &*subcadnue*)

Reemplaza el rango especificado por *inicio* y *final* con *subcadnue*. Por tanto, se modifica la cadena que invoca. Devuelve una referencia a la cadena que invoca.

Ejemplo

He aquí una manera de crear la función **buscar_ign_mayus()**. Utiliza **comp_ign_mayus()** para determinar cuándo dos caracteres son iguales.

```
// Ignora la diferencia entre mayúsculas y minúsculas cuando busca.
// una subcadena. La cadena en que se busca se pasa en cad. La
// subcadena que se buscará se pasa en subcad. Devuelve un iterador
// al principio de la coincidencia o cad.end() si no se encuentra una.
//
// Obsérvese que se usa el algoritmo search() y especifica el
// predicado binario comp_ign_mayus().
string::iterator buscar_ign_mayus(string &cad, const string &subcad) {
  return search(cad.begin(), cad.end(),
                subcad.begin(), subcad.end(),
                comp_ign_mayus);
}
```

Como lo indican los comentarios, **buscar_ign_mayus()** encuentra (independientemente de las diferencias entre mayúsculas y minúsculas) la primera aparición de **subcad** y devuelve un iterador al principio de la secuencia coincidente. Devuelve **cad.end()** si no se encuentra una coincidencia.

He aquí una manera de implementar **buscar_y_reemplazar_ign_mayus()**. Observe que utiliza **buscar_ign_mayus()** para encontrar la subcadena que se reemplazará.

```
// Esta función reemplaza la primera aparición de subcadant con
// subcadnue en la cadena pasada en cad. Devuelve true si ocurre
// un reemplazo, y falso, si no.
//
// Observe que esta función modifica la cadena a la que cad hace
// referencia. Además, nótese que usa buscar_ign_mayus() para encontrar la
// subcadena que se reemplazará.
bool buscar_y_reemplazar_ign_mayus(string &cad, const string &subcadant,
                        const string &subcadnue) {
  string::iterator itrinicio;

  itrinicio = buscar_ign_mayus(cad, subcadant);

  if(itrinicio != cad.end()) {
    cad.replace(itrinicio, itrinicio+subcadant.size(), subcadnue);
    return true;
  }

  return false;
}
```

Esta función reemplaza la primera aparición de **subcadant** con **subcadnue**. Devuelve el valor true si ocurre un reemplazo (es decir, si **cad** contiene **subcadant**) y falso, si no. Como lo indican los comentarios, esta función modifica **cad** en el proceso. Utiliza **buscar_ign_mayus()** para encontrar la primera aparición de **subcadant**. Por tanto, la búsqueda se realiza independientemente de las diferencias entre mayúsculas y minúsculas.

En el siguiente ejemplo se muestran **buscar_ign_mayus()** y **buscar_y_reemplazar_ign_mayus()** en acción:

```
// Implementa búsquedas y búsquedas y reemplazo no sensibles
// a mayúsculas y minúsculas para objetos de cadena.
#include <iostream>
#include <string>
#include <cctype>
#include <algorithm>

using namespace std;

bool comp_ign_mayus(char x, char y);
string::iterator buscar_ign_mayus(string &cad, const string &subcad);
bool buscar_y_reemplazar_ign_mayus(string &cad, const string &subcadant,
                                 const string &subcadnue);

int main()
{
  string cadA("Es una prueba no sensible a may\u00a3sculas y min\u00a3sculas.");
  string cadB("prueba");
```

```
  string cadC("PRUEBA");
  string cadD("pruebas");

  cout << "Primero, se demuestra buscar_ign_mayus().\n";
  cout << "Cadena en que se busca:\n" << cadA << "\n\n";

  cout << "Buscando " << cadB << ". ";
  if(buscar_ign_mayus(cadA, cadB) != cadA.end())
    cout << "Encontrada\n";

  cout << "Buscando " << cadC << ". ";
  if(buscar_ign_mayus(cadA, cadC) != cadA.end())
    cout << "Encontrada\n";

  cout << "Buscando " << cadD << ". ";
  if(buscar_ign_mayus(cadA, cadD) != cadA.end())
    cout << "Encontrada\n";
  else
    cout << "No encontrada\n";

  // Usa el iterador devuelto por buscar_ign_mayus() para
  // desplegar el resto de la cadena.
  cout << "\nEl resto de la cadena tras encontrar 'no':\n";
  string::iterator itr = buscar_ign_mayus(cadA, "no");
  while(itr != cadA.end())
    cout << *itr++;
  cout << "\n\n";

  // Ahora, demuestra la búsqueda y reemplazo.
  cadA = "Alfa Beta Gamma alfa beta gamma";
  cout << "Ahora se demuestra buscar_y_reemplazar_ign_mayus().\n";
  cout << "Cadena que recibe los reemplazos:\n" << cadA << "\n\n";
  cout << "Reemplazando todos los casos de alfa con zeta:\n";
  while(buscar_y_reemplazar_ign_mayus(cadA, "alfa", "zeta"))
    cout << cadA << endl;

  return 0;
}

// Ignora la diferencia entre mayúsculas y minúsculas cuando busca.
// una subcadena. La cadena en que se busca se pasa en cad. La
// subcadena que se buscará se pasa en subcad. Devuelve un iterador
// al principio de la coincidencia o cad.end() si no se encuentra una.
//
// Obsérvese que se usa el algoritmo search() y especifica el
// predicado binario comp_ign_mayus().
string::iterator buscar_ign_mayus(string &cad, const string &subcad) {
  return search(cad.begin(), cad.end(),
                subcad.begin(), subcad.end(),
                comp_ign_mayus);
}

// Ignora la  diferencia entre mayúsculas y minúsculas cuando se compara
// la igualdad entre dos caracteres. Devuelve true si los caracteres
```

```
// son iguales, independientemente de las diferencias entre mayúsculas
// y minúsculas.
bool comp_ign_mayus(char x, char y) {
  return tolower(x) == tolower(y);
}

// Esta función reemplaza la primera aparición de subcadant con
// subcadnue en la cadena pasada en cad. Devuelve true si ocurre
// un reemplazo, y falso, si no.
//
// Observe que esta función modifica la cadena a la que cad hace
// referencia. Además, nótese que usa buscar_ign_mayus() para encontrar la
// subcadena que se reemplazará.
bool buscar_y_reemplazar_ign_mayus(string &cad, const string &subcadant,
                        const string &subcadnue) {
  string::iterator itrinicio;

  itrinicio = buscar_ign_mayus(cad, subcadant);

  if(itrinicio != cad.end()) {
    cad.replace(itrinicio, itrinicio+subcadant.size(), subcadnue);
    return true;
  }

  return false;
}
```

Aquí se muestra la salida:

```
Primero, se demuestra buscar_ign_mayus().
Cadena en que se busca:
Es una prueba no sensible a mayúsculas y minúsculas.

Buscando prueba. Encontrada.
Buscando PRUEBA. Encontrada.
Buscando pruebas. No encontrada.

El resto de la cadena tras encontrar 'no':
no sensible a mayúsculas y minúsculas.

Ahora se demuestra buscar_y_reemplazar_ign_mayus().
Cadena que recibe los reemplazos:
Alfa Beta Gamma alfa beta gamma

Reemplazando todos los casos de alfa con zeta:
zeta Beta Gamma alfa beta gamma
zeta Beta Gamma zeta beta gamma
```

Opciones

Aunque el autor prefiere implementar una búsqueda no sensible a mayúsculas y minúsculas mediante el uso del algoritmo STL **search()** como en esta solución, hay otro método. Puede implementar usted mismo esta función de búsqueda, trabajando carácter tras carácter y tratando de encontrar manualmente una subcadena coincidente. He aquí una manera de hacer esto:

```
// Implementa manualmente buscar_ign_mayus().
// Como la versión original, la cadena de búsqueda se pasa en cad
// y la subcadena que se buscará se pasa en subcad. Devuelve
// un iterador al inicio de la coincidencia o cad.end()
// si no se encuentra una coincidencia.
string::iterator buscar_ign_mayus(string &cad, const string &subcad) {
  string::iterator inicio1, encontrada_en;
  string::const_iterator inicio2;

  // Si la cadena coincidente es nula, devuelve un iterador al
  // principio de cad.
  if(subcad.begin() == subcad.end()) return cad.begin();

  inicio1 = encontrada_en = cad.begin();
  while(inicio1 != cad.end()) {
    inicio2 = subcad.begin();
    while(tolower(*inicio1) == tolower(*inicio2)) {
      ++inicio1;
      ++inicio2;
      if(inicio2 == subcad.end()) return encontrada_en;
      if(inicio1 == cad.end()) return cad.end();
    }

    ++encontrada_en;
    inicio1 = encontrada_en;
  }
  return cad.end();
}
```

Como verá, el método manual incluye mucho más código. Es más, el desarrollo y la prueba de esta función toma más tiempo que el uso del algoritmo STL **search()**. Por último, no se hizo un intento de optimizar el código anterior. La optimización también toma una cantidad importante de tiempo. Por esto, casi siempre son preferibles los algoritmos STL a los métodos "caseros".

La función **tolower()** convierte caracteres con base en la configuración regional de idioma. Para comparar caracteres para una configuración diferente, puede usar la versión de **tolower()** que se declara dentro de **<locale>**.

Aunque no hay ventaja en hacerlo, también es posible convertir cada carácter en la cadena a mayúsculas (en lugar de minúsculas) para eliminar las diferencias entre mayúsculas y minúsculas. Esto se hace mediante la función **toupper()**, que se muestra aquí:

int toupper(int *car*)

Funciona igual que **tolower()**, con la excepción de que convierte caracteres a mayúsculas.

Convierta un objeto string en una cadena terminada en un carácter nulo

Componentes clave		
Encabezado	**Clases**	**Funciones**
<string>	string	const char *c_str() const

La clase **string** proporciona mecanismos fáciles que toman una cadena terminada en un carácter nulo y la convierten en un objeto **string**. Por ejemplo, puede construir una cadena que se inicializa con una cadena terminada en un carácter nulo. También puede asignar una de estas cadenas a un objeto **string**. Por desgracia, el procedimiento inverso no es muy fácil. La razón es que la cadena terminada en un carácter nulo no es un tipo de datos, sino una convención. Esto significa que no puede inicializar este tipo de cadena con una **string** ni asignar una **string** a un apuntador **char ***, por ejemplo. Sin embargo, **string** proporciona la función **c_str()** que convierte un objeto **string** en una cadena terminada en un carácter nulo. En esta solución se muestra el proceso.

Paso a paso

Para obtener una cadena terminada en un carácter nulo que contenga la misma secuencia de caracteres que si lo encapsulara un objeto **string**, siga estos pasos:

1. Cree una matriz de **char** que sea lo suficientemente grande como para contener los caracteres contenidos en el objeto **string**, además del terminador de carácter nulo. Puede tratarse de una matriz declarada estáticamente o una que se asigne dinámicamente mediante **new**.
2. Para obtener un apuntador a una cadena terminada en un carácter nulo que corresponda a la cadena contenida en un objeto **string**, llame a **c_str()**.
3. Copie la cadena terminada en un carácter nulo obtenida en el paso 2 en la matriz creada en el paso 1.

Análisis

Para obtener una representación de una cadena terminada en un carácter nulo de la secuencia de carácter almacenada en el objeto **string**, llame a **c_str()**, que se muestra aquí:

```
const char *c_str() const
```

Aunque no es necesario que la secuencia de caracteres en una **string** termine en un carácter nulo, el apuntador devuelto por una llamada a **c_str()** señalará siempre a una matriz de cadena terminada en un carácter nulo que contiene la misma secuencia. Sin embargo, tome en cuenta que el apuntador devuelto es **const**. Por tanto, no puede usarse para modificar la cadena. Más aún, este apuntador es válido sólo hasta que se llama a una función miembro que no es **const** en el mismo objeto **string**. Como resultado, por lo general querrá copiar la cadena terminada en un carácter nulo en otra matriz.

Ejemplo

En el siguiente ejemplo se muestra cómo convertir un objeto de cadena en una cadena terminada en un carácter nulo:

```
// Convierte un objeto string en una cadena terminada en un carácter nulo.
#include <iostream>
#include <string>
#include <cstring>

using namespace std;

int main()
{
  string cad("Se trata de una prueba.");
  char ccad[80];

  cout << "La cadena original:\n";
  cout << cad << "\n\n";

  // Obtiene un apuntador a la cadena.
  const char *p = cad.c_str();

  cout << "La versi\u00a2n de la cadena terminada en un car\u00a0cter nulo:\n";
  cout << p << "\n\n";


  // Copia la cadena en una matriz asignada estáticamente.
  //
  // Primero, confirma que la matriz tenga la longitud necesaria
  // para contener la cadena.
  if(sizeof(ccad) < cad.size() + 1) {
    cout << "La matriz es demasiado peque\u00a8a para contener la cadena.\n";
    return 0;
  }
  strcpy(ccad, p);
  cout << "La cadena copiada en ccad:\n" << ccad << "\n\n";


  // Luego, copia la cadena en una matriz asignada dinámicamente.
  try {
    // Asigna dinámicamente la matriz.
    char *p2 = new char[cad.size()+1];

    // Copia la cadena en la matriz.
    strcpy(p2, cad.c_str());

    cout << "La cadena tras copiarse en una matriz asignada din\u00a0micamente:\n";
    cout << p2 << endl;

    delete [] p2;
  } catch(bad_alloc ba) {
    cout << "Fall\u00a2 la asignaci\u00a2n\n";
    return 1;
  }

  return 0;
}
```

Aquí se muestra la salida:

```
La cadena original:
Se trata de una prueba.

La versión de la cadena terminada en un carácter nulo:
Se trata de una prueba.

La cadena copiada en ccad:
Se trata de una prueba.

La cadena tras copiarse en una matriz asignada dinámicamente:
Se trata de una prueba.
```

Opciones

Como se explicó, la función **c_str()** devuelve un apuntador a una matriz terminada en un carácter nulo de **char**. Si sólo necesita acceder a los caracteres que integran la secuencia encapsulada por una cadena, sin el terminador de carácter nulo, entonces puede usar la función **data()**. Devuelve un apuntador a una matriz de **char** que contiene los caracteres, pero esa matriz no termina en un carácter nulo. Aquí se muestra:

const char *data() const

Debido a que se devuelve un apuntador **const**, no puede usarlo para modificar los caracteres de la matriz. Si quiere modificar la secuencia de caracteres, cópiela en otra matriz.

Aunque el apuntador devuelto por **c_str()** es **const**, es posible sobreescribir esto al usar **const_cast**, como se muestra aquí:

```
char *p = const_cast<char *> (cad.c_str());
```

Después de que se ejecuta esta instrucción, sería posible modificar la secuencia de caracteres a la que señala **p**. *Sin embargo, ¡no se recomienda hacer esto!* Al cambiar la secuencia de caracteres controlada por un objeto **string** desde código exterior al objeto podría causar fácilmente que el objeto se corrompa, lo que podría llevar a que el programa deje de funcionar o produzca una brecha de seguridad. Por tanto, los cambios al objeto **string** siempre deben tomar lugar mediante funciones miembro de **string**. Nunca debe tratar de cambiar la secuencia mediante un apuntador devuelto por **c_str()** o **data()**. Si ve una construcción como ésta, debe considerarlo código no válido y dar pasos para remediar la situación.

Implemente la resta para objetos string

Componentes clave		
Encabezado	**Clases**	**Funciones**
<string>	string	string &erase(size_type *ind* = 0, size_type *long* = npos) size_type find(const string &*cad*, size_type *ind* = 0) const

Como sabe, el operador + está sobrecargado por objetos de tipo **string** y une dos cadenas y devuelve el resultado. Sin embargo, el operador – no está sobrecargado para **string**. Algunos programadores encuentran esto un poco sorpresivo porque, intuitivamente, se esperaría que se use el operador – para eliminar una subcadena de una cadena, como se ilustra con esta secuencia:

```
string cadA("uno dos tres");
string cadB;
string = cad-"dos";
```

En este punto, esperaría que **cadB** contenga la secuencia "uno tres", que es la secuencia original con la palabra "dos" eliminada. Por supuesto, esto no es posible empleando sólo los operadores definidos para **string** por la biblioteca estándar, porque la resta no es uno de ellos. Por fortuna, es muy fácil remediar esta situación, como se muestra en esta solución.

Para dar soporte a resta de subcadenas, se implementan los operadores – y –= para objetos de tipo **string**. Cada uno elimina la primera aparición de la cadena a la izquierda de la cadena de la derecha. En el caso de –, se devuelve el resultado pero no se modifica ninguno de los dos operandos. Para –=, la subcadena se elimina del operando de la izquierda. Por tanto, se modifica el operando de la izquierda.

Paso a paso

Para sobrecargar **operator–()** para objetos de tipo **string** se requieren estos pasos:

1. Cree una versión de **operator–()** que tenga el siguiente prototipo:

   ```
   string operator-(const string &izq, const string &der);
   ```

   Cuando una cadena se resta de otra, la cadena de la izquierda será **izq** y la de la derecha será **der**.
2. Dentro de **operator–()**, cree una cadena que contendrá el resultado de la resta, e inicialice esa cadena con la secuencia de caracteres de **izq**.
3. Use **find()** para encontrar la primera aparición de **der** en la cadena resultante.
4. Si se encuentra una subcadena resultante, use **erase()** para eliminar la subcadena de la cadena de resultado.
5. Devuelva la cadena resultante.

Para sobrecargar **operator–=()** para objetos de tipo **string** se requieren estos pasos:

1. Cree una versión de **operator–=()** que tenga el siguiente prototipo:

   ```
   string operator-= ( string &izq, const string &der);
   ```

   Aquí, la cadena de la izquierda será **izq** y la de la derecha será **der**. Más aún, **izq** recibirá el resultado de la resta.
2. Dentro de **operator–()**, use **find()** para encontrar la primera aparición de **der** en la cadena a la que hace referencia con **izq**.
4. Si se encuentra una subcadena resultante, use **erase()** para eliminar la subcadena de **izq**. Esto da como resultado que se modifique la cadena en **izq**.
5. Devuelva **izq**.

Análisis

Cuando los operadores binarios están sobrecargados por funciones que no son miembros, el operando de la izquierda siempre se pasa en el primer parámetro y el de la derecha en el segundo. Por tanto, dada una función **operator–()** con este prototipo:

```
string operator-(const string &izq, const string &der);
```

la expresión

```
cadA – cadB
```

causa que se pase a **izq** una referencia a **cadA** y a **de** una a **cadB**. Más aún, dada una función **operator-=()** con este prototipo:

```
string operator-=(string &izq, const string &der);
```

La instrucción

```
cadA –= cadB
```

causa que se pase a **izq** una referencia a **cadA** y a **der** una a **cadB**.

Aunque no hay un mecanismo que lo imponga, por lo general es mejor sobrecargar operadores de una manera consistente con su significado y sus efectos normales. Por tanto, cuando un operador binario como – está sobrecargado, se devuelve el resultado pero no se modifica ninguno de los dos operandos. Esto sigue el uso normal de la – en expresiones como 10-3. En este caso, el resultado es 7, pero no se modifica ni 10 ni 3. Por supuesto, la situación es diferente para la operación –=. En este caso, el operando de la izquierda recibe la salida de la operación. Por tanto, un **operator–=()** sobrecargado modifica el operando de la izquierda. En esta solución se siguen estas convenciones.

El proceso real de eliminar la primera aparición de una subcadena es muy fácil, y sólo incluye dos pasos principales. En primer lugar, se llama a la función **find()** de **string** para localizar el inicio de la primera coincidencia. La función **find()** está detallada en *Busque un objeto **string***, pero he aquí un breve resumen. La función **find()** tiene varias formas. La que se usa aquí es:

size_type find(const string &*cad*, size_type *ind* = 0) const

Devuelve el índice de la primera aparición de *cad* dentro de la cadena que invoca. La búsqueda empieza en el índice especificado por *ind*. Se devuelve **npos** si no se encuentra una coincidencia.

Suponiendo que se encuentre una coincidencia, se elimina la subcadena al llamar a **erase()**. Esta función se analiza en *Realice operaciones básicas en cadenas terminadas en un carácter nulo*. He aquí una rápida recapitulación. La función **erase()** tiene tres formas. Aquí se muestra la usada en esta solución:

string &erase(size_type *ind* = 0, size_type *long* = npos)

Empezando en *ind*, elimina *long* caracteres a partir de la cadena que invoca. Devuelve una referencia a la cadena que invoca.

Cuando se implementa **operator–()**, ninguno de los operandos debe modificarse. Por tanto, debe usarse una cadena temporal que contendrá el resultado de la resta. Inicialice esta cadena con la secuencia de caracteres en el operando de la izquierda. Luego, elimine la subcadena especificada por el operando de la derecha. Por último, devuelva el resultado.

Cuando se implementa **operator–=()**, el operando de la izquierda debe contener el resultado de la resta. Por tanto, se elimina la subcadena especificada por el operando de la derecha a partir de la cadena a la que se hace referencia con el operando de la izquierda. Aunque este último contiene el resultado, también debe devolver la cadena resultante. Esto permite que el operador –= se use como parte de una expresión más larga.

Ejemplo

He aquí una manera de implementar **operator–()** y **operator–=()** para objetos de tipo **string**:

```
// Sobrecarga - (resta) para objetos string de modo que elimina
// la primera aparición de la subcadena de la izquierda a partir
// de la cadena de la derecha y devuelve el resultado. Ninguno
// de los operandos se modifica. Si no se encuentra la subcadena
// el resultado contiene la misma cadena que el operando izquierdo.
string operator-(const string &izq, const string &der) {
  string::size_type i;
  string resultado(izq);

  i = resultado.find(der);
  if(i != string::npos)
    resultado.erase(i, der.size());

  return resultado;
}

// Sobrecarga -= para objetos string. Elimina la primera aparición
// de la subcadena de la derecha de la cadena de la izquierda. Por
// tanto, se modifica la cadena a la que considera en la izquierda.
// También se devuelve la cadena resultante.
string operator-=(string &izq, const string &der) {
  string::size_type i;

  i = izq.find(der);
  if(i != string::npos)
    izq.erase(i, der.size());

  return izq;
}
```

En el siguiente ejemplo se muestran estos operadores en acción:

```
// Implementa operator-() y operator-=() para cadenas.
#include <iostream>
#include <string>

using namespace std;

string operator-(const string &izq, const string &der);
string operator-=(string &izq, const string &der);

int main()
{
  string cad("S\u00a1, esto es una prueba.");
```

```
  string res_cad;

  cout << "Contenido de cad: " << cad << "\n\n";

  // Resta "es" de cad y coloca el resultado en res_cad.
  res_cad =  cad - "es";
  cout << "Resultado de cad - \"es\": " << res_cad << "\n\n";

  // Usa -= para restar "es" de res_cad. Esto regresa el
  // resultado a res_cad.
  res_cad -= "es";
  cout << "Resultado de res_cad -= \"es\": " << res_cad << "\n\n";
  cout << "Se muestra de nuevo cad: " << cad
       << "\nNote que cad ha quedado sin cambio por las "
       << "operaciones anteriores." << "\n\n";

  cout << "Algunos ejemplos adicionales:\n\n";

  // Trata de restar "xyz". Esto no provoca cambios.
  res_cad =  cad - "xyz";
  cout << "Resultado de cad - \"xyz\": " << res_cad << "\n\n";

  // Elimina los últimos tres caracteres de cad.
  res_cad =  cad - "ba.";
  cout << "Resultado de cad - \"ba.\": " << res_cad << "\n\n";

  // Elimina una cadena nula, lo que no produce cambios.
  res_cad =  cad - "";
  cout << "Resultado de cad - \"\": " << res_cad << "\n\n";

  return 0;
}

// Sobrecarga - (resta) para objetos string de modo que elimina
// la primera aparición de la subcadena de la izquierda a partir
// de la cadena de la derecha y devuelve el resultado. Ninguno
// de los operandos se modifica. Si no se encuentra la subcadena
// el resultado contiene la misma cadena que el operando izquierdo.
string operator-(const string &izq, const string &der) {
  string::size_type i;
  string resultado(izq);

  i = resultado.find(der);
  if(i != string::npos)
    resultado.erase(i, der.size());

  return resultado;
}

// Sobrecarga -= para objetos string. Elimina la primera aparición
// de la subcadena de la derecha de la cadena de la izquierda. Por
// tanto, se modifica la cadena a la que considera en la izquierda.
// También se devuelve la cadena resultante.
string operator-=(string &izq, const string &der) {
```

```
  string::size_type i;

  i = izq.find(der);
  if(i != string::npos)
    izq.erase(i, der.size());

  return izq;
}
```

Aquí se muestra la salida:

```
Contenido de cad: Sí, esto es una prueba.

Resultado de cad - "es": Sí, to es una prueba.

Resultado de res_cad -= "es": Sí, to  una prueba.

Se muestra de nuevo cad: Sí, esto es una prueba.
Note que cad ha quedado sin cambio por las operaciones anteriores.

Algunos ejemplos adicionales:

Resultado de cad - "xyz": Sí, esto es una prueba.

Resultado de cad - "ba.": Sí, esto es una prue

Resultado de cad - "": Sí, esto es una prueba.
```

Opciones

Las versiones de **operator–()** y **operator–=()** descritas en la solución sólo eliminan la primera aparición de la subcadena en la derecha de la cadena de la izquierda. Sin embargo, con un poco de trabajo, puede cambiar su operación para que elimine todas las apariciones de la subcadena. He aquí una manera de hacerlo:

```
// Sobrecarga - (resta) para objetos string de modo que elimina
// TODAS las apariciones de la subcadena en la izquierda a partir
// de la cadena de la derecha. Se devuelve el resultado. No se
// modifica ninguno de los operandos.
string operator-(const string &izq, const string &der) {
  string::size_type i;
  string resultado(izq);

  if(der != "") {
    do {
      i = resultado.find(der);
      if(i != string::npos)
        resultado.erase(i, der.size());
    } while(i != string::npos);
  }

  return resultado;
}
```

```
// Sobrecarga -= para objetos string de modo que elimina
// TODAS las apariciones de la subcadena en la derecha a partir
// de la cadena de la izquierda. El resultado se incluye en la
// cadena señalada por el operando de la izquierda. Por tanto,
// se modifica el operando de la izquierda. También se devuelve.
// la cadena resultante.
string operator-=(string &izq, const string &der) {
  string::size_type i;

  if(der != "") {
    do {
      i = izq.find(der);
      if(i != string::npos)
        izq.erase(i, der.size());
    } while(i != string::npos);
  }

  return izq;
}
```

Otra opción que tal vez le resulte útil en algunos casos es implementar la resta de cadenas para que opere de manera independiente de diferencias entre mayúsculas y minúsculas. Para ello, utilice el método descrito en *Cree una búsqueda no sensible a mayúsculas y minúsculas y funciones de búsqueda y reemplazo para objetos **string*** para realizar una búsqueda no sensible a mayúsculas y minúsculas para encontrar la subcadena que se eliminará.

3

CAPÍTULO

Trabajo con contenedores STL

Éste es el primero de dos capítulos que presentan soluciones que usan la biblioteca de plantillas estándar (STL, Standard Template Library). Se necesitan dos capítulos porque la STL es una parte extraordinariamente grande e importante de C++. No sólo proporciona soluciones preelaboradas a algunos de los problemas de programación más desafiantes, también redefine la manera en que se pueden enfrentar muchas tareas comunes. Por ejemplo, en lugar de tener que proporcionar su propio código para una lista vinculada, puede usar la clase **list** de STL. Si su programa necesita asociar una clave con un valor y proveer un medio para encontrar ese valor dada la clave, puede usar la clase **map**. Debido a que STL proporciona implementaciones sólidas, depuradas, de los "motores de datos" de uso más común, puede usar uno sin importar lo que necesite, sin dedicar el tiempo necesario ni afrontar el problema de desarrollar los propios.

Este capítulo empieza con una revisión general de la STL, y luego presenta soluciones que demuestran el núcleo de la STL: sus contenedores. En el proceso, muestra la manera en que los iteradores se utilizan para acceder y recorrer en ciclo el contenido de un contenedor. En el siguiente capítulo se muestra cómo usar algoritmos y varios otros componentes de la STL.

He aquí las soluciones contenidas en este capítulo:

- Técnicas básicas de contenedor de secuencias
- Use **vector**
- Use **deque**
- Use **list**
- Use los adaptadores de contenedor de secuencias: **snack**, **queue** y **priority_queue**
- Almacene en un contenedor objetos definidos por el usuario
- Técnicas básicas de contenedor asociativo
- Use **map**
- Use **multimap**
- Use **set** y **multiset**

NOTA *Para conocer una descripción a fondo de STL, consulte el libro* Programming from the Ground Up. *Gran parte de la revisión general y las descripciones de este capítulo están adaptadas de ese trabajo. La STL también recibe amplia cobertura en el libro* C++: The Complete Reference. *Ambos libros son de Herb Schildt.*

Revisión general de STL

En esencia, la biblioteca de plantillas estándar es un conjunto complejo de clases y funciones de plantilla que implementa muchas estructuras de datos y algoritmos populares y de uso común. Por ejemplo, incluye soporte para vectores, listas, colas y pilas. También proporciona muchos algoritmos (como de ordenamiento, búsqueda y combinación) que operan en ellos. Debido a que STL está construido a partir de clases y funciones de plantillas, las estructuras de datos y los algoritmos pueden aplicarse a casi cualquier tipo de datos. Esto es, por supuesto, parte de su poder.

STL está organizado alrededor de tres elementos básicos: *contenedores*, *algoritmos* e *iteradores*. Para ponerlo en palabras simples, los algoritmos actúan como contenedores mediante iteradores. Más que otra cosa, el diseño y la implementación de estas características determinan la naturaleza de STL. Además de contenedores, algoritmos e iteradores, STL depende de otros diversos elementos estándar para soporte: *asignadores*, *adaptadores*, *objetos de función*, *predicados*, *adhesivos* y *negadores*.

Contenedores

Como su nombre lo indica, un contenedor es un objeto que puede contener otros objetos. Hay varios tipos diferentes de contenedores. Por ejemplo, la clase **vector** define una matriz dinámica, **deque** crea una cola de doble extremo, y **list** proporciona una lista vinculada. A estos contenedores se les denomina *contenedores de secuencia* porque, en terminología de STL, una secuencia es una lista lineal. La STL también define *contenedores asociativos*, que permiten recuperación eficiente de valores basados en claves. Por tanto, los contenedores asociativos almacenan pares clave/valor. Un **map** es un ejemplo. Almacena pares clave/valor en que cada clave es única. Esto facilita la recuperación de un valor específico dada su clave.

Algoritmos

Los algoritmos actúan como contenedores. Entre sus capacidades se incluyen inicialización, ordenamiento, búsqueda, combinación, reemplazo y transformación de contenido de un contenedor. Muchos algoritmos operan en un *rango* de elementos dentro de un contenedor.

Iteradores

Los iteradores son objetos que actúan, más o menos, como apuntadores. Le dan la capacidad de recorrer en ciclo el contenido de un contenedor de manera muy parecida a como se usaría uno para recorrer de la misma forma una matriz. Hay cinco tipos de iteradores:

Iterador	Acceso permitido
Acceso aleatorio	Almacena y recupera valores. Los elementos pueden accederse de manera aleatoria.
Bidireccional	Almacena y recupera valores. Movimiento directo e inverso.
Directo	Almacena y recupera valores. Sólo se mueve hacia adelante.
Entrada	Recupera, pero no almacena valores. Sólo se mueve hacia adelante.
Salida	Almacena, pero no recupera valores. Sólo se mueve hacia adelante.

En general, un iterador que tiene mayores capacidades de acceso puede usarse en lugar de uno que tiene menores opciones. Por ejemplo, un iterador directo puede usarse en lugar de uno de entrada.

Los iteradores se manejan como apuntadores. Puede aumentarlos o disminuirlos. Puede aplicar los operadores * y –>. Los iteradores se declaran usando el tipo **iterator** definido por diversos contenedores.

La STL también da soporte a varios iteradores. Los iteradores inversos son bidireccionales o de acceso aleatorio y recorren una secuencia en dirección inversa. Por tanto, si un iterador inverso señala al final de una secuencia, el aumento de ese iterador causará que señale a un elemento antes del final.

Todos los iteradores deben dar soporte a los tipos de operadores de apuntador permitidos en esa categoría. Por ejemplo, una clase de iterador de entrada debe dar soporte a –>, ++, *, == y !=. Más aún, el operador * no puede usarse para asignar un valor. En contraste, un iterador de acceso aleatorio debe dar soporte a –>, +, ++, –, – –, *, <, >, <=, >=, –=, +=, ==, != y []. Además, el * debe permitir asignación. A continuación se muestran los operadores con soporte para cada tipo:

Iterador	Operaciones soportadas
Acceso aleatorio	*, –>, =, +, –, ++, – –, [],<, >, <=, >=, – =, +=, ==, !=
Bidireccional	*, –>, =, ++, – –, ==, !=
Directo	*, –>, =, ++, ==, !=
Entrada	*,–>, =, ++, ==, !=
Salida	*, =, ++

Cuando se hace referencia a varios tipos de iteradores en descripciones de plantillas, en este libro se usarán los siguientes términos:

Término	Representa
BiIter	Iterador bidireccional
ForIter	Iterador directo
InIter	Iterador de entrada
OutIter	Iterador de salida
RandIter	Iterador de acceso aleatorio

Asignadores

Cada contenedor tiene definido un *asignador.* Los asignadores administran la asignación de memoria a un contenedor. El asignador predeterminado es un objeto de clase **allocator**, pero pueden definirse los propios, si es necesario, para aplicaciones especializadas. Para casi todos los usos, basta con el asignador predeterminado.

Objetos de función

Los *objetos de función* son instancias de clases que definen **operator()**. Hay varios objetos de funciones predefinidos, como **less()**, **greater()**, **plus()**, **minus()**, **multiplies()** y **divides()**. Tal vez el objeto de función de uso más extenso sea **less()**, que determina cuando un objeto es menos que otro. Los objetos de función pueden usarse en lugar de apuntadores de función en los algoritmos STL. Los objetos de función aumentan la eficiencia de algunos tipos de operaciones y proporcionan soporte para ciertas operaciones que, de otra manera, no sería posible usando sólo un apuntador a función.

Adaptadores

En el sentido más general, un *adaptador* transforma una cosa en otra. Hay adaptadores de contenedor, de iterador y de función. Un ejemplo de adaptador de contenedor es **queue**, que adapta el contenedor **deque** para usar como una cola estándar.

Predicados

Varios de los algoritmos y contenedores usan un tipo especial de función llamada *predicado*. Hay dos variaciones de predicados: unarios y binarios. Un predicado unario toma un argumento. Un predicado binario tiene dos argumentos. Estas funciones devuelven resultados true/false, pero usted define las condiciones precisas que hacen que devuelva uno de estos valores. En este libro, cuando se requiere un predicado unario, se indicará usando el tipo **UnPred**. Cuando se necesita un predicado binario, se usará el tipo **BinPred**. En un predicado binario, los argumentos siempre están en el orden *primero*, *segundo*. Para los predicados unarios y binarios, los argumentos contendrán valores de tipo de objetos que están almacenados en el contenedor.

Algunos algoritmos usan un tipo especial de predicado binario que compara dos elementos. Las *funciones de comparación* devuelven true si su primer argumento es menos que el segundo. En este libro, las funciones de comparación se indicarán usando el tipo **Comp**.

Adhesivos y negadores

Otras dos entidades que pueblan las STL son los adhesivos y los negadores. Un *adhesivo* une un argumento a un objeto de función. Un *negador* devuelve el complemento de un predicado. Ambos aumentan la versatilidad de la STL.

La clase de contenedor

En el núcleo de la STL se encuentran sus contenedores. Se muestran en la tabla 3-1. También se muestran los encabezados necesarios para usar cada contenedor. Como podría esperarse, cada contenedor tiene diferentes capacidades y atributos.

Los contenedores se implementan usando clases de plantillas. Por ejemplo, aquí se muestra la especificación de plantilla para el contenedor **deque**. Todos los contenedores usan especificaciones similares:

```
template <class T, class Allocator = allocator<T> > class deque
```

Aquí, el tipo genérico **T** especifica el tipo de objetos contenidos por **deque**. El asignador usado por **deque** se especifica con **Allocator**, que tiene como opción predeterminada la clase asignadora estándar. Para la mayor parte de las aplicaciones, simplemente usará el asignador predeterminado, y eso es lo que se hace en todo el código de este capítulo. Sin embargo, es posible definir su propia clase asignadora si se llega a necesitar un esquema de asignación especial. Si no está familiarizado con los argumentos predeterminados en las plantillas, sólo recuerde que funcionan de manera muy parecida a los argumentos predeterminados en funciones. Si el argumento de tipo genérico no está especificado explícitamente cuando se crea un objeto, entonces se usa el tipo predeterminado.

Contenedor	Descripción	Encabezado requerido
deque	Una cola de doble extremo.	<deque>
list	Una lista lineal.	<list>
map	Almacena pares clave/valor en que cada clave está asociada con un solo valor.	<map>
multimap	Almacena pares clave/valor en que una clave puede estar asociada con dos o más valores.	<multimap>
multiset	Un conjunto en que cada elemento no es necesariamente único.	<multiset>
priority_queue	Una cola con prioridades.	<queue>
queue	Una cola.	<queue>
set	Un conjunto en que cada elemento es único.	<set>
stack	Una pila.	<stack>
vector	Una matriz dinámica.	<vector>

TABLA 3-1 Contenedores definidos por la STL.

Cada clase de contenedor incluye varios **typedef** que crean un conjunto de nombres de tipo estándar. Varios de estos nombres de **typedef** se muestran aquí:

size_type	Algún tipo de entero sin signo.
reference	Una referencia a un elemento.
const_reference	Una referencia **const** a un elemento.
iterator	Un iterador.
const_iterator	Un iterador **const**.
reverse_iterator	Un iterador inverso.
const_reverse_iterator	Un iterador inverso **const**.
value_type	El tipo de valor almacenado en un contenedor. Igual que **T** para los contenedores de secuencia.
allocator_type	El tipo del asignador.
key_type	El tipo de una clave.

Como se mencionó, hay dos amplias categorías de contenedores: de secuencia y asociativas. Los de secuencia son **vector**, **list** y **deque**. Los asociativos son **map**, **multimap**, **set** y **multiset**. Los contenedores de secuencia, como su nombre lo indica, operan en secuencias, que son, en sí, listas lineales de objetos. Los contenedores asociativos operan en listas de claves. De estos últimos, los que implementan mapas operan en pares clave/valor y permiten la recuperación de un valor dada su clave.

A las clases **stack**, **queue** y **priority_queue** se les denomina *adaptadores de contenedor* porque usan (es decir, adaptan) uno de los contenedores de secuencia para que contengan sus elementos. Por tanto, uno de los contenedores de secuencia subraya la funcionalidad proporcionada por **stack**, **queue** y **priority_queue**. Desde la perspectiva del programador, los adaptadores de contenedor se parecen a los otros contenedores y actúan como ellos.

Funcionalidad común

La STL especifica un conjunto de requisitos que todos los contenedores deben satisfacer. Al especificar una funcionalidad común, STL asegura que los algoritmos pueden actuar sobre todos los contenedores, y que todos los contenedores pueden usarse de una manera bien entendida y consistente que es independiente de los detalles de cada implementación de contenedor. Esta es otra de las fortalezas importantes de STL.

Todos los contenedores deben dar soporte al operador de asignación. También deben dar soporte a todos los operadores lógicos. En otras palabras, todos los contenedores deben dar soporte a estos operadores:

=, ==, <, <=, !=, >, >=

Todos los contenedores deben proporcionar un constructor que cree un contenedor vacío y una copia del constructor. Deben proveer un destructor que libere toda la memoria usada por el contenedor y llamar al destructor para todos los elementos del contenedor.

Todos los contenedores también deben dar soporte a iteradores. Entre otras ventajas, esto asegura que los algoritmos puedan operar en todos los adaptadores.

Todos los contenedores deben proveer las siguientes funciones:

iterator begin()	Devuelve un iterador al primer elemento del contenedor.
const_iterator begin() const	Devuelve un iterador **const** al primer elemento del contenedor.
bool empty() const	Devuelve true si el contenedor está vacío.
iterator end()	Devuelve un iterador a uno después del último elemento en el contenedor.
const_iterator end() const	Devuelve un iterador **const** a uno después del último elemento en el contenedor.
size_type max_size() const	Devuelve el número máximo de elementos que puede incluir el contenedor.
size_type size() const	Devuelve el número de elementos almacenados en el contenedor.
void swap(*TipoContenedor* c)	Intercambia el contenido de dos contenedores.

A un contenedor que da soporte a acceso bidireccional a sus elementos se le denomina *contenedor reversible*. Además de los requisitos básicos, un contenedor reversible también debe proporcionar iteradores inversos y las siguientes funciones:

reverse_iterator rbegin()	Devuelve un iterador inverso al último elemento en el contenedor.
const_reverse_iterator rbegin() const	Devuelve un iterador **const** inverso al último elemento en el contenedor.
reverse_iterator rend()	Devuelve un iterador inverso a uno antes del primer elemento del contenedor.
const_reverse_iterator rend() const	Devuelve un iterador **const** inverso a uno antes del primer elemento del contenedor.

Requisitos de contenedor de secuencias

Además de la funcionalidad común a todos los contenedores, un contenedor de secuencias agrega las siguientes funciones:

void clear()	Elimina todos los elementos del contenedor.
iterator erase(iterator *i*)	Elimina los elementos señalados por *i*. Devuelve un iterador al elemento después del eliminado.
iterator erase(iterator *inicio*, iterator *final*)	Elimina elementos en el rango especificado por *inicio* y *final*. Devuelve un iterador al elemento que sigue el último elemento eliminado.
iterator insert(iterator *i*, const T &*val*)	Inserta *val* inmediatamente antes del elemento especificado por *i*. Devuelve un iterador al elemento.
void iterator insert(iterator *i*, size_type *num*, const T &*val*)	Inserta *num* copias de *val* inmediatamente antes del elemento especificado por *i.*
template <class InIter> void insert(iterator *i*, InIter *inicio*, InIter *final*)	Inserta la secuencia definida por *inicio* y *final* inmediatamente antes del elemento especificado por *i*.

La STL define un conjunto de funciones para contenedores de secuencia que son opcionales, pero que se implementan con frecuencia. Aquí se muestran:

reference at(size_type *ind*)	Devuelve una referencia a un elemento especificado por *ind*.
const_reference at(size_type *ind*) const	Devuelve una referencia **const** a un elemento especificado por *ind*.
reference back()	Devuelve una referencia al último elemento del contenedor.
const_reference back() const	Devuelve una referencia **const** al último elemento del contenedor.
reference front()	Devuelve una referencia al primer elemento del contenedor.
const_reference front() const	Devuelve una referencia **const** al primer elemento del contenedor.
reference operator[](size_type *ind*)	Devuelve una referencia al elemento especificado por *ind*.
const_reference operator[](size_type *ind*) const	Devuelve una referencia **const** al elemento especificado por *ind*.
void pop_back()	Elimina el último elemento del contenedor.
void pop_front()	Elimina el primer elemento del contenedor.
void push_back(const T &*val*)	Agrega un elemento con el valor especificado por *val* al final del contenedor.
void push_front(const T &*val*)	Agrega un elemento con el valor especificado por *val* al inicio del contenedor.

Los contenedores de secuencia también deben proporcionar constructores que permiten que un contenedor se inicialice mediante elementos especificados por un par de iteradores o con un número específico de un elemento determinado. Por supuesto, un contenedor de secuencias tiene la libertad de proporcionar funcionalidad adicional.

Requisitos de contenedores asociativos

Además de la funcionalidad requerida de todos los contenedores, los asociativos tienen varios otros requisitos. En primer lugar, todos deben dar soporte a las siguientes funciones:

void clear()	Elimina todos los elementos del contenedor.
size_type count(const key_type &*c*) const	Devuelve el número de veces que *c* se presenta en el contenedor.
void erase(iterator *i*)	Elimina los elementos señalados por *i*.
void erase(iterator *inicio*, iterator *final*)	Elimina los elementos en el rango *inicio* y *final*.
size_type erase(const key_type &*c*)	Elimina los elementos que tienen claves con el valor *c*. Devuelve el número de elementos que se han eliminado.
pair<iterator, iterator> equal_range(const key_type &*c*)	Devuelve un par de iteradores que señalan a los límites superior e inferior en el contenedor para la clave especificada.
pair<const_iterator, const_iterator> equal_range(const key_type &*c*) const	Devuelve un par de iteradores **const** que señalan a los límites superior e inferior en el contenedor para la clave especificada.
iterator find(const key_type &*c*)	Devuelve un iterador a la clave especificada. Si no se encuentra, entonces se devuelve un iterador al final del contenedor.
const_iterator find(const key_type &*c*) const	Devuelve un iterador **const** a la clave especificada. Si no se encuentra, entonces se devuelve un iterador al final del contenedor.
pair<iterator, bool> insert(const value_type &*val*)	Inserta *val* en el contenedor. Si éste requiere claves únicas, entonces *val* sólo se inserta si aún no existe. Si los elementos están insertados, se devuelve **pair<iterator, true>**. De otra manera, se devuelve **pair<iterator, false>**.
iterator insert(iterator *inicio*, const value_type &*val*)	Inserta *val*. La búsqueda del punto de inserción apropiado empieza en el elemento especificado por *inicio*. En el caso de contenedores que requieren claves únicas, los elementos se insertan sólo si aún no existen. Se devuelve un iterador al elemento.
template <class InIter> void insert(InIter *inicio*, InIter *final*)	Inserta un rango de elementos. En el caso de contenedores que requieren claves únicas, los elementos se insertan sólo si aún no existen.
key_compare key_comp() const	Devuelve el objeto de función que compara dos claves.
iterator lower_bound(const key_type &*c*)	Devuelve el iterador al primer elemento con una clave igual o mayor que *c*.
const_iterator lower_bound(const key_type &*c*) const	Devuelve el iterador **const** al primer elemento con una clave igual o mayor que *c*.
iterator upper_bound(const key_type &*c*)	Devuelve un iterador al primer elemento con una clave mayor que *c*.
const_iterator upper_bound(const key_type &*c*) const	Devuelve un iterador **const** al primer elemento con una clave mayor que *c*.
value_compare value_comp() const	Devuelve el objeto de función que compara los dos valores.

Observe que algunas de las funciones devuelven un objeto **pair**. Se trata de una clase que encapsula dos objetos. En el caso de contenedores asociativos que son mapas, **value_type** representa un **pair** que encapsula una clave y un valor. La clase **pair** se explica de manera detallada en *Técnicas básicas de contenedor asociativo*.

Los contenedores asociativos deben proporcionar constructores que permitan que los elementos especificados por un par de iteradores inicialicen un contenedor. Deben también dar soporte a constructores que le permitan especificar las funciones de comparación usadas para comparar dos claves. Por supuesto, un contenedor asociativo tiene la libertad de proporcionar funcionalidad adicional.

Problemas de rendimiento

Hay otro aspecto importante relacionado con las STL que se añade a su capacidad y su aplicabilidad general: las garantías de rendimiento. Aunque un fabricante de compiladores tiene la libertad de implementar los mecanismos usados por otro contenedor y algoritmo por su cuenta, todas las implementaciones deben adecuarse a las garantías de rendimiento especificadas por el STL. Se definen las siguientes categorías generales de rendimiento:

constante
lineal
logarítmica

Como diferentes contenedores almacenan su contenido de manera diferente, tendrán garantías de rendimiento distintas. Por ejemplo, la inserción en la parte media de un **vector** ocupa tiempo lineal. En contraste, la inserción en una **list** toma tiempo constante. Diferentes algoritmos podrían comportarse de manera diferente. Por ejemplo, el algoritmo **sort()** se ejecuta de manera proporcional a N log N, pero **find()** lo hace en tiempo lineal.

En algunos casos, se dirá que una operación toma *tiempo constante amortizado*. Este término se emplea para describir una situación en que una operación suele tomar tiempo constante, pero en ocasiones requiere más. (Por ejemplo, inserciones al final del vector suelen ocurrir en tiempo constante, pero si debe asignarse más memoria, entonces la inserción requiere tiempo lineal.) Si la operación más larga es lo suficientemente rara, entonces puede considerarse como amortizada a través de varias operaciones más cortas.

En general, la especificación STL requiere que los contenedores y algoritmos se implementen usando técnicas que aseguran (hablando de manera general) un rendimiento óptimo del motor en tiempo de ejecución. Esto es importante porque le garantiza al programador que los bloques de construcción de STL cumplan con un cierto nivel de eficiencia, sin importar qué implementación de STL se esté usando. Sin esta garantía, el rendimiento del código basado en STL dependería por completo de cada implementación individual y podría variar ampliamente.

Técnicas básicas de contenedor de secuencias

Componentes clave		
Encabezados	**Clases**	**Funciones**
<vector>	vector	iterator begin() void clear() bool empty() const iterator end() iterator erase(iterator *i*) iterator insert(iterator *i*, const T &*val*) reverse_iterator rbegin() reverse_iterator rend() size_type size() const void swap(vector<T, Allocator> &*ob*)
<Vector>		template <class T, class Allocator> bool operator==(const vector<T, Allocator> &*izqsup* const vector<T, Allocator> &*dersup*) template <class T, class Allocator> bool operator<(const vector<T, Allocator> &*izqsup* const vector<T, Allocator> &*dersup*) template <class T>, class Allocator> bool operator>(const vector<T, Allocator> &*izqsup*, const vector<T, Allocator> &*dersup*)

Todos los contenedores de secuencias comparten una funcionalidad común. Por ejemplo, todos le permiten agregar elementos al contenedor, eliminar elementos de él o recorrerlo en ciclo mediante un iterador. Todos dan soporte al operador de asignación y los operadores lógicos, y todos están construidos de la misma manera. En esta solución se describe esta funcionalidad común, mostrando las técnicas básicas que aplican a todos los contenedores de secuencias.

En esta solución se muestra cómo:

- Crear un contenedor de secuencias.
- Agregar elementos al contenedor.
- Determinar el tamaño del contenedor.
- Usar un iterador para recorrer en ciclo el contenedor.

- Asignar un contenedor a otro.
- Determinar cuando un contenedor es equivalente a otro.
- Eliminar elementos de un contenedor.
- Intercambiar los elementos de un contenedor con otro.
- Determinar si un contenedor está vacío.

En esta solución se usa la clase de contenedor **vector**, pero sólo se emplean los métodos comunes a los contenedores de secuencias. Por tanto, pueden aplicarse los mismos principios generales a cualquier tipo de contenedor de secuencias.

Paso a paso

Para crear y usar un contenedor de secuencias se requieren estos pasos:

1. Cree una instancia del contenedor deseado. En esta solución, se usa **vector**, pero puede sustituirse con cualquier otro contenedor de secuencias.
2. Agregue elementos al contenedor al llamar a **insert()**.
3. Obtenga varios elementos en el contenedor al llamar a **size()**.
4. Determine si el contenedor está vacío (es decir, no contiene elementos) al llamar a **empty()**.
5. Elimine elementos del contenedor al llamar a **erase()**.
6. Elimine todos los elementos de un contenedor al llamar a **clear()**.
7. Obtenga un iterador al principio de la secuencia al llamar a **begin()**. Obtenga un iterador a uno después del final de la secuencia al llamar a **end()**.
8. En el caso de contenedores de secuencias reversibles, obtenga un iterador inverso al final de la secuencia, al llamar a **rbegin()**. Obtenga un iterador inverso a uno antes del inicio de la secuencia al llamar a **rend()**.
9. Recorra en ciclo los elementos del contenedor mediante un iterador.
10. Intercambie el contenido de un contenedor con otro mediante **swap()**.
11. Determine cuando un contenedor es igual, menor que, o mayor que otro.

Análisis

Aunque la operación interna de STL es muy compleja, su empleo en realidad es muy fácil. En muchos aspectos, la parte más difícil de su uso está en decidir cuál tipo de contenedor se debe usar. Cada uno ofrece ciertos beneficios y requiere ciertas compensaciones. Por ejemplo, **vector** es muy bueno cuando se requiere un objeto de acceso aleatorio, tipo matriz, y no se harán demasiadas inserciones o eliminaciones. Una **list** ofrece inserción y eliminación a bajo costo, pero a cambio de búsquedas lentas. Una cola de doble extremo tiene soporte por parte de **deque**. En esta solución se usa **vector** para demostrar las operaciones básicas de contenedores de secuencias, pero el programa funcionará con **list** o **deque**. Ésta es una de las ventajas más importantes de STL; todos los contenedores de secuencias dan soporte a un nivel básico de funcionalidad común.

La especificación de plantilla para **vector** se muestra a continuación:

template <class T, class Allocator = allocator<T> > class **vector**

Aquí, **T** es el tipo de datos que se habrán de almacenar y **Allocator** especifica el asignador, que es, como opción predeterminada, el estándar. Para usar **vector**, debe incluir el encabezado **<vector>**.

La clase **vector** da soporte a varios constructores. Los dos usados en esta solución son los necesarios para todos los contenedores de secuencias. Se muestran a continuación:

explicit vector(const Allocator %*asign* = Allocator())
vector(const vector<T, Allocator> &*ob*)

La primera forma construye un vector vacío. La segunda forma es un constructor de copia de **vector**.

Después de que se ha creado un contenedor, pueden agregársele objetos. Una manera de hacer esto y que funciona para todos los contenedores de secuencias es llamar a **insert()**. Todos los contenedores de secuencias dan soporte, por lo menos, a tres versiones de **insert()**. Ésta es la usada aquí:

iterator insert(iterator *i*, const T &*val*)

Inserta *val* en el contenedor que invoca en el punto especificado por *i*. Devuelve un iterador al elemento insertado. Un contenedor de secuencias crecerá automáticamente a medida que se necesita cuando se le agreguen elementos.

Puede eliminar uno o más elementos de un contenedor de secuencias al llamar a **erase()**. Tiene por lo menos dos formas. La usada en esta solución se muestra a continuación:

iterator erase(iterator *i*)

Elimina el elemento al que señala *i*. Devuelve un iterador al elemento después del eliminado. Para eliminar todos los elementos en un contenedor, llame a **clear()**. Aquí se muestra:

void clear()

Puede determinar el número de elementos en un contenedor al llamar a **size()**. Para determinar si un contenedor está vacío, llame a **empty()**. Ambas funciones se muestran a continuación:

bool empty() const
size_type size() const

Puede obtener un iterador al principio de la secuencia al llamar a **begin()**. Un iterador a uno después del último elemento en la secuencia se obtiene al llamar a **end()**. Aquí se muestran estas funciones:

iterator begin()
iterator end()

También hay versiones **const** de estas funciones.

Para declarar una variable que se usará como iterador, debe especificar el tipo de iterador del contenedor. Por ejemplo, esto declara un iterador que puede apuntar a elementos dentro de un **vector<double>**:

```
vector<double>::iterator itr;
```

Es útil destacar que **end()** *no* devuelve un iterador que señala al último elemento del contenedor. En cambio, devuelve un iterador que señala a *uno después* del último elemento. Por tanto, el último elemento de un contenedor es señalado por **end()**–1. Esta característica le permite escribir algoritmos muy eficientes que recorren en ciclo todos los elementos de un contenedor, incluido el último, usando un iterador. Cuando éste tiene el mismo valor que el devuelto por **end()**, sabrá que se ha tenido acceso a todos los elementos. Por ejemplo, he aquí un bucle que recorre todos los elementos de un contenedor de secuencias llamado **cont**:

```
for(itr = cont.begin(); itr != cont.end(); ++itr) // ...
```

El bucle se ejecuta hasta que **itr** es igual a **cont.end()**. Por tanto, todos los elementos de **cont** tendrán que procesarse.

Como ya se explicó, un contenedor reversible es uno en que los elementos pueden recorrerse en orden inverso (de atrás hacia adelante). Todos los contenedores de secuencias integrados son reversibles. Para un contenedor reversible, puede obtener un iterador inverso al final de la secuencia al llamar a **rbegin()**. Un iterador a uno antes del primer elemento en la secuencia se obtiene al llamar a **rend()**. Aquí se muestran estas funciones:

reverse_iterator rbegin()
reverse_iterator rend()

También hay versiones **const** de estas funciones. Un iterador inverso se declara como un iterador regular. Por ejemplo,

```
vector<double>::reverse_iterator ritr;
```

Puede usar un iterador inverso para recorrer en ciclo un vector en orden inverso. Por ejemplo, dado un iterador inverso llamado **ritr**, he aquí un bucle que recorre todos los elementos en un contenedor de secuencias reversible llamado **cont** de atrás hacia adelante:

```
for(ritr = cont.rbegin(); ritr != cont.rend(); ++ritr) // ...
```

El iterador inverso **ritr** empieza en el elemento señalado por **rbegin()**, que es el último elemento de la secuencia. Se ejecuta hasta que es igual a **rend()**, que señala a un elemento que está uno antes del inicio de la secuencia. (En ocasiones resulta útil considerar a **rbegin()** y **rend()** como iteradores que regresan al inicio y el final de una secuencia invertida.) Cada vez que se aumenta un iterador inverso, señala al elemento anterior. Cada vez que se reduce, señala al siguiente elemento.

El contenido de dos contenedores de secuencia puede intercambiarse al llamar a **swap()**. He aquí la manera en que se define para **vector**:

void swap(vector<T, Allocator> &*ob*)

El contenido del contenedor que invoca se intercambia con el especificado por *ob*.

Ejemplo

En el siguiente ejemplo se demuestran las operaciones básicas de contenedor de secuencias:

```
// Demuestra las operaciones básicas de contenedor de secuencias.
//
// En este ejemplo se usa vector, pero puede aplicarse la misma
// técnica a cualquier contenedor de secuencias.
```

```
#include <iostream>
#include <vector>

using namespace std;

void mostrar(const char *msg, vector<char> vect);

int main() {
  // Declara un vector vacío que puede contener objetos char.
  vector<char> v;

  // Declara un iterador a un vector<char>.
  vector<char>::iterator itr;

  // Obtiene un iterador al principio de v.
  itr = v.begin();

  // Inserta caracteres en v. Se devuelve un iterador al
  // objeto insertado.
  itr = v.insert(itr, 'A');
  itr = v.insert(itr, 'B');
  v.insert(itr, 'C');

  // Despliega el contenido de v.
  mostrar("El contenido de v: ", v);

  // Declara un iterador inverso.
  vector<char>::reverse_iterator ritr;

  // Usa un iterador inverso para mostrar el contenido de v en reversa.
  cout << "Se muestra v en reversa: ";
  for(ritr = v.rbegin(); ritr != v.rend(); ++ritr)
    cout << *ritr << " ";
  cout << "\n\n";

  // Crea otro vector que es el mismo que el primero.
  vector<char> v2(v);
  mostrar("El contenido de v2: ",v2);
  cout << "\n";

  // Muestra el tamaño de v, que es el número de elementos
  // contenidos por v.
  cout << "El tama\u00a4o de v es " << v.size() << "\n\n";

  // Compara dos contenedores.
  if(v == v2) cout << "v y v2 son equivalentes.\n\n";

  // Inserta más caracteres en v y v2. Esta vez,
  // se insertan al final.
  cout << "Se insertan caracteres adicionales en v y v2.\n";
  v.insert(v.end(), 'D');
  v.insert(v.end(), 'E');
```

```
  v2.insert(v2.end(), 'X');
  mostrar("El contenido de v: ", v);
  mostrar("El contenido de v2: ", v2);
  cout << "\n";

  // Determina si v es menos que v2. Se trata de
  // una comparación lexicográfica. Por ello, el
  // primer elemento no coincidente del contenedor determina
  // cuál contenedor es menos que otro.
  if(v < v2) cout << "v es menos que v2.\n\n";

  // Ahora, inserta Z al inicio de v.
  cout << "Se inserta Z al inicio de v.\n";
  v.insert(v.begin(), 'Z');
  mostrar("El contenido de v: ", v);
  cout << "\n";

  // Ahora, compara v con v2 una vez más.
  if(v > v2) cout << "Ahora, v es mayor que v2.\n\n";

  // Elimina el primer elemento de v2.
  v2.erase(v2.begin());
  mostrar("v2 tras eliminar el primer elemento: ", v2);
  cout << "\n";

  // Crea otro vector.
  vector<char> v3;
  v3.insert(v3.end(), 'X');
  v3.insert(v3.end(), 'Y');
  v3.insert(v3.end(), 'Z');
  mostrar("El contenido de v3: ", v3);
  cout << "\n";

  // Intercambia el contenido de v y v3.
  cout << "Se intercambian v y v3.\n";
  v.swap(v3);
  mostrar("El contenido de v: ", v);
  mostrar("El contenido de v3: ", v3);
  cout << "\n";

  // Limpia v.
  v.clear();
  if(v.empty()) cout << "v ahora est\u00a0 vac\u00a1o.";

  return 0;
}

// Despliega el contenido de un vector<char> al usar
// un iterador.
void mostrar(const char *msg, vector<char> vect) {
  vector<char>::iterator itr;

  cout << msg;
  for(itr=vect.begin(); itr != vect.end(); ++itr)
```

```
    cout << *itr << " ";
  cout << "\n";
}
```

Aquí se muestra la salida:

```
El contenido de v: C B A
Se muestra v en reversa: A B C

El contenido de v2: C B A

El tamaño de v es 3

v y v2 son equivalentes.

Se insertan caracteres adicionales en v y v2.
El contenido de v: C B A D E
El contenido de v2: C B A X

v es menos que v2.

Se inserta Z al inicio de v.
El contenido de v: Z C B A D E

Ahora, v es mayor que v2.

v2 tras eliminar el primer elemento: B A X

El contenido de v3: X Y Z

Se intercambian v y v3.
El contenido de v: X Y Z
El contenido de v3: Z C B A D E

v ahora está vacío.
```

Aunque gran parte del programa se explica por sí solo, hay varios puntos de interés que merecen una revisión de cerca. En primer lugar, observe que no se especifica algún asignador cuando se declaran los contenedores del programa (**v**, **v2** y **v3**). Como se explicó, en casi todos los usos de STL, la opción correcta es el asignador predeterminado.

A continuación, observe cómo el iterador **itr** se declara en esta instrucción:

```
vector<char>::iterator itr;
```

Esto declara un iterador que puede usarse con objetos de tipo **vector<char>**. Cada clase de contenedor crea un **typedef** para **iterator**. Los iteradores a otros tipos de vectores u otros contenedores se declaran de la misma manera general. Por ejemplo:

```
vector<double>::iterator itrA;
deque<string>::iterator itrB;
```

Aquí, **itrA** es un iterador que puede usarse en contenedores **vector<double>** e **itrB** aplica a contenedores de tipo **deque<string>**. En general, debe declarar un iterador de una manera que coincida

con los tipos de contenedor y de objetos contenidos en él. Lo mismo es válido para los iteradores inversos.

A continuación, se obtiene un iterador al principio del contenedor al llamar a **begin()**, y luego el siguiente conjunto de llamadas a **insert()** pone elementos en **v**:

```
itr = v.insert(itr, 'A');
itr = v.insert(itr, 'B');
v.insert(itr, 'C');
```

Cada llamada inserta el valor que se encuentra inmediatamente antes del elemento señalado por el iterador y pasado en **itr**. Se devuelve un iterador al elemento insertado. Por tanto, estas tres llamadas causan que **v** contenga la secuencia CBA.

Ahora, revise la función **mostrar()**. Se usa para desplegar el contenido de un **vector<char>**. Preste especial atención al siguiente bucle:

```
for(itr=vect.begin(); itr != vect.end(); ++itr)
  cout << *itr << " ";
```

Recorre en ciclo el vector pasado a **vect**, empezando con el primer elemento y deteniéndose cuando se ha encontrado el último. Recuerde que **end()** devuelve un iterador que señala a un elemento después del final del contenedor. Por tanto, cuando **itr** es igual a **vect.end()**, se ha alcanzado el final del contenedor. Estos tipos de bucles son muy comunes cuando se trabaja con STL. Además, observe cómo se vuelve a hacer referencia a **itr** mediante el operador * casi de la misma manera en que se haría con un apuntador. En general, los iteradores funcionan como apuntadores y se manejan, en esencia, de la misma manera.

A continuación, en **main()**, observe cómo el iterador inverso **ritr** se usa para recorrer en ciclo el contenido de **v** en orden inverso. Un iterador inverso funciona de manera parecida a uno normal, excepto que accede a los elementos del contenedor en orden inverso.

Ahora, observe cómo se comparan dos contenedores mediante el uso de los operadores == y <. En el caso de contenedores de secuencias, las comparaciones son lexicográficas y se aplican a los elementos. Aunque el término "lexicográfico" significa "orden del diccionario", su significado se generaliza a la manera en que se relaciona con STL. En el caso de comparaciones entre contenedores, dos de éstos son iguales si contienen el mismo número de elementos, en el mismo orden, y todos los elementos correspondientes son iguales. De otra manera, el resultado de la comparación lexicográfica se basa en los primeros elementos que no coinciden. Por ejemplo, dadas estas dos secuencias:

sec1: 7, 8, 9
sec2: 7, 8, 11

sec1 es menos que sec2 porque la primera diferencia es entre 9 y 11, y 9 es menor que 11. Debido a que la comparación es lexicográfica, sec1 es todavía menos que sec2, aunque la longitud de sec1 se aumente a 7, 8, 9, 10, 11, 12. Los primeros elementos no coincidentes (en este caso, 9 y 11) determinan el resultado.

Opciones

Además de la versión de **insert()** usada en esta solución, todos los contenedores de secuencias dan soporte a las dos formas mostradas aquí:

void insert(iterator *i*, size_type *num*, const T &*val*)
template <class InIter> void insert(iterator *i*, InIter *inicio*, InIter *final*)

La primera forma inserta *num* copias de *val* inmediatamente antes del elemento especificado por *i*. La segunda forma inserta la secuencia que se ejecuta desde *inicio* hasta *final*-1 inmediatamente antes del elemento especificado por *i*. Observe que *inicio* y *final* no necesitan estar señalados dentro del contenedor que invoca. Por tanto, esta forma puede usarse para insertar elementos de un contenedor en otro. Más aún, no es necesario que los contenedores sean del mismo tipo. Siempre y cuando los elementos sean compatibles, puede insertar elementos de **deque** en **list**, por ejemplo.

Hay una segunda forma de **erase()** que tiene soporte en todos los contenedores de secuencias. Aquí se muestra:

iterator erase(iterator *inicio*, iterator *final*)

Esta versión elimina elementos en el rango *inicio* a *final*-1 y devuelve un iterador al elemento después del último elemento eliminado.

Además de los operadores **==**, **<** y **>**, todos los contenedores de secuencias dan soporte a los operadores lógicos **<=**, **>=** y **!=**.

Puede encontrar el número máximo de elementos que un contenedor puede incluir al llamar a **max_size()**, que se muestra aquí:

size_type max_size() const

Debe comprender que el tamaño máximo variará, dependiendo del tipo de datos que incluye el contenedor. Además, diferentes tipos de contenedores pueden tener (y probablemente tendrán) diferentes capacidades máximas.

Como se mencionó, el ejemplo anterior funciona para todos los contenedores de secuencias. Para probar esto, trate de sustituir **vector** con **deque** o **list**. Como verá, el programa produce el mismo resultado. Por supuesto, la elección del contenedor apropiado es una parte importante del uso correcto de la STL. Recuerde que diferentes contenedores tendrán diferentes garantías de rendimiento. Por ejemplo, la inserción de un elemento en medio de una **deque** toma tiempo lineal. La inserción en una **list** toma tiempo constante. La inserción en la parte media de un **vector** usa tiempo lineal, pero la misma al final puede ocurrir en tiempo constante (si no se requiere una reasignación). En general, si no hay una razón poderosa para elegir un contenedor sobre otro, **vector** suele ser la mejor elección porque implementa lo que es, en esencia, una matriz dinámica (consulte *Use* ***vector***).

En algunos casos, querrá usar uno de los adaptadores de contenedores de secuencias, como **queue**, **stack** o **priority_queue**, que proporciona una funcionalidad específica que desee. Por ejemplo, si quiere que un contenedor implemente una pila clásica, entonces use **stack**. Para colas de un solo extremo, use **queue**. Para una cola que está ordenada de acuerdo con una prioridad, use **priority_queue**.

Use vector

Componentes clave		
Encabezados	**Clases**	**Funciones**
<vector>	vector	template <class InIter> void assign(InIter *inicio*, InIter *final*) reference at(size_type *i*) reference back() size_type capacity() const reference front() reference operator()(size_type *i*) void pop_back void push_back(const T $*val*) void reserve(size_type *num*) void resize(size_type *num*, T *val* = T())

En esta solución se demuestra **vector**, que es probablemente el contenedor de secuencias de uso más extendido porque implementa una matriz dinámica. A diferencia de una estática, cuyas dimensiones se fijan en tiempo de compilación, una matriz dinámica puede crecer de acuerdo con las necesidades durante la ejecución del programa. Esto hace que **vector** resulte una excelente opción para situaciones en que necesita una matriz, pero no sabe por anticipado el tamaño que debe tener. Aunque la matriz creada por **vector** es dinámica, aún puede accederse a sus elementos empleando el operador de subíndice de matriz normal []. Esto facilita la colocación de **vector** en situaciones en que, de otra manera, se requeriría una matriz.

NOTA *El eje de esta solución está en los atributos y las características de* ***vector*** *que lo hacen único. Consulte* Técnicas básicas de contenedor de secuencias *para conocer información que aplica a todos los contenedores de secuencias.*

Paso a paso

Para usar **vector** se requieren los siguientes pasos:

1. Cree una instancia de **vector** del tipo deseado y el tamaño inicial.
2. Asigne u obtenga valores para los elementos mediante el operador de subíndice.
3. Use la función **at()** como una opción al operador de subíndice.
4. Agregue elementos al vector usando **insert()** o **push_back()**.
5. Elimine elementos del final al llamar a **pop_back()**.
6. Obtenga una referencia al primer elemento del vector al llamar a **front()**.

7. Obtenga una referencia al último elemento del vector al llamar a **back()**.
8. Asigne un rango de elementos a un vector al llamar a **assign()**.
9. Para obtener la capacidad actual de un vector, llame a **capacity()**. Para especificar una capacidad, llame a **reserve()**.
10. Para cambiar el tamaño de un vector, llame a **resize()**.

Análisis

Aquí se muestra la especificación de la plantilla para **vector**:

```
template <class T, class Allocator=allocator<T> > class vector
```

Aquí, **T** es el tipo de datos que se están almacenando y **Allocator** especifica el asignador, que es, como opción predeterminada, el asignador estándar. Para usar **vector**, debe incluir el encabezado **<vector>**.

He aquí los constructores de **vector**:

```
explicit vector(const Allocator &asign = Allocator())
explicit vector(size_type num, const T &val = T (),
                const Allocator &asign = Allocator())
vector(const vector<T, Allocator> &ob)
template <class InIter> vector(InIter inicio, InIter final,
          const Allocator &asign = Allocator())
```

La primera forma construye un vector vacío. La segunda, uno que tiene *num* elementos con el valor *val*. La tercera es un constructor de copia de **vector**. La cuarta construye un vector que contiene los elementos en el rango *inicio* a *final*-1. El asignador usado por el vector se especifica con *asign*, que suele permitirse como opción predeterminada.

La clase **vector** da soporte a iteradores de acceso aleatorio, y el [] está sobrecargado. Esto permite que se indice un objeto **vector** como una matriz.

La clase **vector** implementa todas las funciones y operaciones de contenedor de secuencias necesarias, como **erase()**, **insert()**, **swap()** y los operadores lógicos. También proporciona todas las funciones necesarias para un contenedor reversible. Brinda casi todas las funciones opcionales de contenedor de secuencias. Las únicas de estas funciones que no implementa son **push_front()** y **pop_front()**.

Los elementos dentro de un vector pueden accederse de dos maneras. En primer lugar, y lo más conveniente, es mediante el uso del operador de subíndice []. Aquí se muestra:

```
reference operator[](size_type i)
```

Devuelve una referencia al elemento del índice especificado por *i*. El tipo **reference** es un **typedef** para **T &**. (También se proporciona una versión **const** de la función que devuelve una **const_reference**.) Este operador puede utilizarse para establecer u obtener el valor en un índice especificado. Por supuesto, el índice que especifique debe estar dentro del rango actual del vector. Como en las matrices, la indización empieza en cero.

Otra manera de acceder a los elementos en un vector consiste en usar el método **at()**. Aquí se muestra:

```
reference at(size_type i)
```

Devuelve una referencia a un elemento en el índice especificado por *i*. (También se proporciona una versión **const** de la función que devuelve una **const_reference**.) Esta referencia puede emplearse para obtener el valor en un índice especificado. Por supuesto, el índice que especifique debe estar dentro del rango actual del **vector**. Como el operador [], la indización usando **at()** también empieza en cero.

Aunque el uso del operador [] es más conveniente, la función **at()** ofrece un beneficio. Si se hace un intento por acceder a un elemento que está fuera de los límites actuales de **vector**, **at()** lanzará una excepción **out_of_range**. Por tanto, proporciona comprobación de límites. Lo que no hace [].

Aunque todos los vectores tienen un tamaño inicial (que puede ser cero), es posible aumentarlo al agregar elementos al vector. Hay dos maneras fáciles de hacer esto: insertar elementos empleando la función **insert()** y agregar elementos al final al llamar a **push_back()**. La función **insert()** se describe en *Técnicas básicas de contenedor de secuencias* y ya no se trata más aquí. A continuación se muestra la función **push_back()**:

void push_back(const T &*val*)

Agrega un elemento con el valor especificado por *val* al final del vector. El tamaño de éste se aumenta automáticamente para acomodar la adición.

El complemento de **push_back()** es **pop_back()**. Elimina un elemento del final del vector. Se muestra a continuación:

void pop_back()

Después de que se ejecuta **pop_back()**, el tamaño del vector se reduce en uno.

Puede obtener una referencia al último elemento del vector al llamar a **back()**. Se devuelve una referencia al primer elemento mediante **front()**. Aquí se muestran estas funciones:

reference back()
reference front()

La clase **vector** también proporciona versiones **const** de estas funciones.

El tipo de iterador proporcionado por **vector** es de acceso aleatorio. Esto significa que puede agregarse un valor entero al iterador, o restársele a éste, lo que permite que el iterador señale a cualquier elemento arbitrario dentro del contenedor. También permite que un iterador recorra un vector en dirección directa o inversa. La clase **vector** define dos tipos de iterador: directo o inverso. Los iteradores directos son objetos de tipo **iterator** o **const_iterator**, los inversos son de tipo **reverse_iterator** o **const_reverse_iterator**.

Se obtiene un iterador directo al inicio de un vector al llamar a **begin()**, y uno al final se obtiene al llamar a **end()**. Un iterador inverso al final del vector se obtiene al llamar a **rbegin()**, y uno a uno antes del inicio se obtiene con **rend()**. Estas funciones y el procedimiento básico requerido para recorrer en ciclo un contenedor de secuencias se describen en *Técnicas básicas de contenedor de secuencias*.

Puede asignar un nuevo conjunto de valores a un vector al usar la función **assign()**. Tiene dos formas. La usada en esta solución se muestra a continuación:

template <class InIter> void assign(InIter *inicio*, InIter *final*)

Reemplaza todo el contenido del vector que invoca con los valores especificados en el rango *inicio* a *final*-1. Observe que *inicio* y *final* pueden ser cualquier tipo de iterador de entrada. Esto significa

que puede usar **assign()** para asignar valores de otro vector o cualquier otro tipo de contenedor. La única regla es que los valores deben ser compatibles con el objeto que invoca.

Todos los vectores se crean con una *capacidad* inicial. Éste es el número de elementos que puede contener el vector antes de que se necesite asignar más memoria. Puede obtener la capacidad actual al llamar a **capacity()**, que se muestra aquí:

size_type capacity() const

Es importante que no se confunda capacidad con tamaño. El tamaño de un vector, que está disponible al llamar a la función de contenedor estándar **size()**, es el número de elementos que contiene actualmente. La capacidad es cuánto puede contener antes de que ocurra una reasignación.

Puede reservar memoria para un número específico de elementos al llamar a **reserve()**, que se muestra aquí:

void reserve(size_type *num*)

La función **reserve()** reserva memoria al menos por el número de elementos especificado en *num*. En otras palabras, establece la capacidad del **vector** que invoca igual o mayor que *num*. (Por tanto, un compilador tiene la libertad de ampliar la capacidad para obtener mayor eficiencia.) Debido a que el aumento de la capacidad puede causar una reasignación de la memoria, podría invalidar cualquier apuntador o referencia a elementos dentro de un vector. Si sabe de antemano que un vector contendrá un número específico de elementos, entonces el uso de **reserve()** evitará reasignaciones innecesarias que cuestan mucho tiempo.

Tiene la opción de cambiar el tamaño de un vector al llamar a **resize()**, que se muestra aquí:

void resize(size_type *num*, T *val* = T())

Establece el tamaño del vector al especificado por *num*. Si el tamaño se aumenta, entonces los elementos con el valor especificado por *val* se agregan al final. Observe que *val* corresponde al valor predeterminado de **T**. Si disminuye el tamaño del vector, entonces los elementos se eliminan del final.

La clase **vector** tiene las siguientes características de rendimiento. La inserción o eliminación de elementos al final de un vector se presenta en tiempo constante amortizado. Cuando ocurren al principio o en medio, las inserciones o eliminaciones tienen lugar en tiempo lineal. Como se acaba de explicar, es posible reservar espacio adicional en un vector al usar la función **reserve()**. Al asignar previamente memoria adicional, evitará que ocurran reasignaciones. Por tanto, si administra de manera correcta sus vectores, la mayor parte de las inserciones pueden ocurrir en tiempo constante.

El acceso a un elemento mediante el operador de subíndice toma lugar en tiempo constante. En general, el acceso a elementos en un vector es más rápido de lo que sería con cualquier otro contenedor de secuencias definido por STL. Por esto es por lo que **vector** se usa para matrices dinámicas.

En todos los casos, cuando ocurre una inserción, ya no serán válidas las referencias y los iteradores a elementos después del punto de inserción. Sin embargo, en algunos casos, como cuando se agrega el elemento al final mediante una llamada a **push_back()**, es probable que no sean válidas todas las referencias e iteradores a elementos. Esta situación se presenta sólo si es necesario que el vector asigne más memoria. En este caso, ocurre una *reasignación*, y el contenido del vector puede moverse a una nueva ubicación. Si el vector se mueve físicamente, ya no serán válidos los iteradores y las referencias previos. Por tanto, para todos los fines prácticos, es mejor suponer que las referencias y los iteradores no son válidos después de las inserciones. Cuando se elimina un elemento de un vector, ya no son válidos los iteradores y las referencias a elementos que están después del punto de borrado.

Ejemplo

En el siguiente ejemplo se muestra **vector** en acción:

```
// Demuestra vector.

#include <iostream>
#include <vector>

using namespace std;

void mostrar(const char *msg, vector<int> vect);

int main() {

  // Declara un vector que tiene una capacidad inicial de 10.
  vector<int> v(10);

  // Asigna algunos valores a sus elementos. Obsérvese que se
  // hace mediante la sintaxis de subíndice de matriz estándar.
  // Tómese nota de que el número de elementos en el vector
  // se obtiene al llamar a size().
  for(unsigned i=0; i < v.size(); ++i) v[i] = i*i;

  mostrar("El contenido de v: ", v);

  // Calcula el promedio de los valores. Una vez más,
  // observe el uso del operador de subíndice.
  int sum = 0;
  for(unsigned i=0; i < v.size(); ++i) sum += v[i];
  double avg = sum / v.size();
  cout << "El promedio de los elementos es " << avg << "\n\n";

  // Agrega elementos al final de v.
  v.push_back(100);
  v.push_back(121);

  mostrar("v tras incluir elementos al final: ", v);
  cout << endl;

  // Ahora usa pop_back() para eliminar un elemento.
  v.pop_back();
  mostrar("v tras usar back-pop con un elemento: ", v);
  cout << endl;

  cout << "El primero y \u00a3ltimo elemento de v como"
       << " lo indican begin() y end()-1:\n"
       << *v.begin() << ", " << *(v.end()-1) << "\n\n";

  cout << "El primero y \u00a3ltimo elemento de v como"
       << " lo indican rbegin() y rend()-1:\n"
       << *v.rbegin() << ", " << *(v.rend()-1) << "\n\n";

  // Declara un iterador a un vector<int>.
  vector<int>::iterator itr;
```

```
    // Ahora, declara un iterador inverso a un vector<int>
    vector<int>::reverse_iterator ritr;

    // Recorre en ciclo v en dirección directa usando un iterador.
    cout << "Se aplica un bucle al vector en direcci\u00a2n directa:\n";
    for(itr = v.begin(); itr != v.end(); ++itr)
      cout << *itr << " ";
    cout << "\n\n";
    cout << "Ahora, se usa un iterador inverso para aplicar un bucle"
         << " en direcci\u00a2n inversa:\n";

    // Recorre v en ciclo en dirección inversa utilizando un iterador inverso.
    for(ritr = v.rbegin(); ritr != v.rend(); ++ritr)
      cout << *ritr << " ";
    cout << "\n\n";

    // Crea otro vector que contiene un subrango de v.
    vector<int> v2(v.begin()+2, v.end()-4);

    // Despliega el contenido de v2 usando un iterador.
    mostrar("v2 contiene un subrango de v: ", v2);
    cout << endl;

    // Cambia los valores de algunos de los elementos de v2.
    v2[1] = 100;
    v2[2] = 88;
    v2[4] = 99;
    mostrar("Tras las asignaciones, v2 ahora contiene: ", v2);
    cout << endl;

    // Crea un vector vacío y luego le asigna una
    // secuencia que es la inversa de v.
    vector<int> v3;
    v3.assign(v.rbegin(), v.rend());
    mostrar("v3 contiene la inversa de v: ", v3);
    cout << endl;

    // Muestra el tamaño y la capacidad de v.
    cout << "El tama\u00a4o de v es " << v.size() << ". La capacidad es "
         << v.capacity() << ".\n";

    // Ahora, cambia el tamaño de v.
    v.resize(20);
    cout << "Tras llamar a resize(20), el tama\u00a4o de v es "
         << v.size() << " y la capacidad es "
         << v.capacity() << ".\n";

     // Ahora, reserva espacio para 50 elementos.
     v.reserve(50);
     cout << "Tras llamar a reserve(50), el tama\u00a4o de v es "
          << v.size() << " y la capacidad es "
          << v.capacity() << ".\n";

     return 0;
}
```

```
// Despliega el contenido de un vector<int>.
void mostrar(const char *msg, vector<int> vect) {
  cout << msg;
  for(unsigned i=0; i < vect.size(); ++i)
    cout << vect[i] << " ";
  cout << "\n";
}
```

Aquí se muestra la salida:

```
El contenido de v: 0 1 4 9 16 25 36 49 64 81
El promedio de los elementos es 28

v tras incluir elementos al final: 0 1 4 9 16 25 36 49 64 81 100 121

v tras usar back-pop con un elemento: 0 1 4 9 16 25 36 49 64 81 100

El primero y último elemento de v como lo indican begin() y end()-1:
0, 100

El primero y último elemento de v como lo indican rbegin() y rend()-1:
100, 0

Se aplica un bucle al vector en dirección directa:
0 1 4 9 16 25 36 49 64 81 100

Ahora, se usa un iterador inverso para aplicar un bucle en dirección inversa:
100 81 64 49 36 25 16 9 4 1 0

v2 contiene un subrango de v: 4 9 16 25 36

Tras las asignaciones, v2 ahora contiene: 4 100 88 25 99

v3 contiene la inversa de v: 100 81 64 49 36 25 16 9 4 1 0

El tamaño de v es 11. La capacidad es 15.
Tras llamar a resize(20), el tamaño de v es 20 y la capacidad es 22.
Tras llamar a reserve(50), el tamaño de v es 20 y la capacidad es 50.
```

Casi todo el programa se explica por sí solo, pero vale la pena analizar más un par de puntos. En primer lugar, tome nota de que el operador de subíndice se utiliza para asignar un valor a un elemento de un vector o para obtener el valor actual de un elemento. Por tanto, funciona de la misma manera que cuando se aplica a una matriz. Un punto clave que se debe comprender es que sólo puede usar subíndices para acceder a un elemento existente. Por ejemplo, en el programa, **v** tiene al principio 10 elementos. Por tanto, no puede asignar, por ejemplo, un valor **v[15]**. Si necesita expandir un vector después de crearlo, debe usar el método **push_back()**, que agrega un valor al final, o el método **insert()**, que puede usarse para insertar uno o más elementos en cualquier lugar de la secuencia.

En segundo lugar, tome nota de que los iteradores inversos se usan en dos lugares: primero, para recorrer en ciclo un vector en dirección inversa y, después, para llamar a **assign()**, con el fin de asignar a **v3** una secuencia que es la inversa de la de **v**. Es este segundo uso el más interesante. Al emplear un iterador inverso, es posible obtener una secuencia invertida en un paso, en lugar de

los dos que se necesitarían si la secuencia se copiara primero como está y luego se invirtiera. En ocasiones, los operadores inversos pueden mejorar operaciones que, de otra manera, serían complejas.

Opciones

Hay otra forma de **assign()** que le permite asignar un valor a un vector. Se muestra aquí:

void assign(size_type *num*, const T& *val*)

Esta versión elimina todos los elementos anteriormente contenidos por el vector y luego asigna *num* copias de *val* al vector. Esta versión de **assign()** es útil cuando quiere reinicializar un vector a un valor conocido, por ejemplo.

El contenedor **vector** *no* almacena elementos en orden. Sin embargo, es posible ordenar un vector al usar el algoritmo **sort()**. Consulte *Ordene un contenedor*, en el capítulo 4.

En algunos casos, el contenedor **deque** es una buena opción a **vector**. Tiene capacidades similares, como permitir el acceso a sus elementos mediante el operador de subíndice, pero tiene características de rendimiento diferentes. Consulte *Use* ***deque*** para conocer detalles.

La STL también contiene una especificación de **vector** para valores **bool**: **vector<bool>**. Incluye toda la funcionalidad de **vector** y agrega estos dos miembros:

void flip()	Invierte todos los bits en el vector.
static void swap(reference *i*, reference *j*)	Intercambia los bits especificados por *i* y *j*.

Mediante la especificación para **bool**, **vector** puede empaquetar valores true/false en bits individuales. La especificación **vector<bool>** define una clase llamada **reference**, que se usa para emular una referencia a un bit.

Use deque

Componentes clave		
Encabezados	**Clases**	**Funciones**
<deque>	deque	template <class InIter> void assign(InIter *inicio*, InIter *final*) reference at(size_type *i*) reference back() reference front() reference operator[](size_type *i*) void pop_back() void pop_front() void push_back(const T &*val*) void push_front(const T &*val*) void resize(size_type *num*, T *val* = T())

Tal vez el segundo contenedor de uso más común sea **deque**. Hay dos razones para esto. En primer lugar, **deque** da soporte a todas las funciones opcionales definidas por los contenedores de

secuencias. Esto hace que la STL sea un contenedor con características más completas. En segundo lugar **deque** es el contenedor predeterminado de los adaptadores de contenedor **queue** y **stack**. (El contenedor predeterminado usado por **priority_queue** es **vector**.) Esta solución muestra la manera de poner a **deque** en acción.

NOTA *El eje de esta solución está en los atributos y las características de* ***deque*** *que lo hacen único. Consulte* Técnicas básicas de contenedor de secuencias *para conocer información que aplica a todos los contenedores de secuencias.*

Paso a paso

Para usar un **deque** se requieren estos pasos:

1. Cree una instancia de **deque** del tipo deseado y el tamaño inicial.
2. Asigne u obtenga valores para los elementos mediante el operador de subíndice.
3. Use la función **at()** como una opción del operador de subíndice.
4. Agregue elementos a la cola de dos extremos empleando **insert()**, **push_back()** o **push_front()**.
5. Elimine elementos del final al llamar a **pop_back()**. Elimine elementos del frente al llamar a **pop_front()**.
6. Obtenga una referencia al primer elemento en la cola de dos extremos al llamar a **front()**.
7. Obtenga una referencia al último elemento en la cola de dos extremos al llamar a **back()**.
8. Asigne un rango de elementos a una cola de dos extremos al llamar a **assign()**.
9. Para cambiar el tamaño de la cola de dos extremos, llame a **resize()**.

Análisis

La especificación de plantilla para **deque** es:

```
template <class T, class Allocator = allocator<T> class deque
```

Aquí, **T** es el tipo de datos almacenado en la cola de dos extremos y **Allocator** especifica el asignador, que tiene como opción predeterminada el asignador estándar. Para usar **deque**, debe incluir el encabezado **<deque>**.

He aquí los constructores de **deque**:

```
explicit deque(const Allocator &asign = Allocator())
explicit deque(size_type num, const T &val = T (),
        const Allocator &asign = Allocator())
deque(const deque<T, Allocator> &ob)
template <class InIter> vector(InIter inicio, InIter final,
        const Allocator &asign = Allocator())
```

La primera forma construye una cola de dos extremos vacía. La segunda, una **deque** que tiene *num* elementos con el valor *val*. La tercera, una cola de dos extremos que contiene los mismos elementos que *ob*. Éste es el constructor de copia de **deque**. La cuarta forma construye una cola de dos extremos que contiene los elementos en el rango *inicio* a *final*–1. El asignador usado por la cola de dos extremos está especificado por *asign* y suele permitirse como opción predeterminada.

El contenedor **deque** da soporte a los iteradores de acceso aleatorio, y [] está sobrecargado. Esto significa que un objeto **deque** puede indizarse como una matriz. También significa que una

cola de dos extremos puede recorrerse en direcciones directas e inversas mediante el uso de un iterador.

El contenedor **deque** proporciona todas las funciones de contenedor de secuencias requeridas, incluidas las de un contenedor reversible, y todas las funciones de contenedor de secuencias opcionales. Esto hace que **deque** sea el contenedor de propósito más general.

Aunque **deque** y **vector** tienen diferentes características de rendimiento, ofrecen funcionalidad casi idéntica. Por ejemplo, las funciones de secuencias estándares implementadas por **deque**, como **insert()**, **erase()**, **begin()**, **end()**, **rbegin()**, **rend()**, **operator[]()**, **front()**, **back()**, **push_back()**, etc., funcionan en **deque** de la misma manera que en **vector**. La función **resize()** proporcionada por **deque** también funciona como la proporcionada por **vector**. Debido a que se presenta un análisis detallado de estos métodos estándar en *Use* ***vector***, esos análisis no se duplican aquí. (Sin embargo, tome nota de que **deque** no da soporte a los métodos **capacity()** y **reserve()** definidos por **vector**. No son necesarios para **deque**.)

La clase **deque** da soporte a dos funciones no proporcionadas por **vector**: **push_front()** y **pop_front()**. Se muestran aquí:

void push_front(const T &*val*)
void pop_front()

La función **push_front()** agrega un elemento con el valor especificado por *val* al inicio del contenedor. Éste automáticamente aumenta de tamaño para acomodar la adición. La función **pop_front()** elimina un elemento del inicio del contenedor.

La clase **deque** tiene las siguientes características de rendimiento. Insertar o eliminar elementos del final de un objeto **deque** toma lugar en tiempo constante. Cuando ocurre en el medio, las inserciones o eliminaciones tienen lugar en tiempo lineal. El acceso de un elemento mediante el operador de subíndice tiene lugar en tiempo constante. Debido a que la adición o eliminación de elementos de los extremos de una cola de dos extremos son muy eficientes, estas colas resultan una excelente elección cuando esos tipos de operaciones ocurrirán con frecuencia. La capacidad de hacer adiciones eficientes al inicio de la cola de dos extremos es una de las principales diferencias entre **vector** y **deque**.

Una inserción en el medio de un contenedor **deque** invalida todos los iteradores y las referencias al contenido de ese contenedor. Debido a que **deque** suele implementarse como una matriz dinámica de doble extremo, una inserción implica que los elementos existentes se "dispersarán" para acomodarse a los nuevos elementos. Por tanto, si un iterador está señalando a un elemento antes de una inserción, no hay garantía de que estará señalando al mismo elemento después de la inserción. Lo mismo aplica a las referencias.

Una inserción a la cabeza o la cola de **deque** invalida los iteradores, pero no las referencias. Un borrado en la parte media invalida iteradores y referencias. Un borrado limitado a cualquier extremo sólo invalida a esos iteradores y referencias que señalan a elementos que habrán de borrarse.

Ejemplo

En el siguiente ejemplo se muestra **deque** en acción. Para fines de comparación, se vuelve a trabajar el ejemplo usado para **vector**, sustituyendo **deque** por **vector** en todo el listado. Debido a que **vector** y **deque** proveen características muy similares, gran parte de los dos programas son iguales. Por supuesto, las llamadas a **capacity()** y **reserve()** que se encuentran en la versión de **vector** se han eliminado, porque esas funciones no tienen soporte en **deque**. Además, se han agregado las

funciones **push_front()** y **pop-front()**. Como se explicó, **deque** proporciona estas funciones, pero no lo hace **vector**:

```
// Demuestra deque.

#include <iostream>
#include <deque>

using namespace std;

void show(const char *msg, deque<int> q);

int main() {

  // Declara una deque que tiene una capacidad inicial de 10.
  deque<int> dq(10);

  // Asigna algunos valores a sus elementos. Obsérvese que se
  // hace mediante la sintaxis de subíndice de matriz estándar.
  // Tómese nota de que el número de elementos en deque
  // se obtiene al llamar a size().
  for(unsigned i=0; i < dq.size(); ++i) dq[i] = i*i;

  show("El contenido de dq: ", dq);

  // Calcula el promedio de los valores. Una vez más,
  // observe el uso del operador de subíndice.
  int sum = 0;
  for(unsigned i=0; i < dq.size(); ++i) sum += dq[i];
  double avg = sum / dq.size();
  cout << "El promedio de los elementos es " << avg << "\n\n";

  // Agrega elementos al final de dq.
  dq.push_back(100);
  dq.push_back(121);

  show("dq tras incluir elementos al final: ", dq);
  cout << endl;

   // Ahora usa pop_back() para eliminar un elemento.
  dq.pop_back();
  show("dq tras usar back-pop con un elemento: ", dq);
  cout << endl;

  cout << "El primero y \u00a3ltimo elemento de dq como"
       << " lo indican begin() y end()-1:\n"
       << *dq.begin() << ", " << *(dq.end()-1) << "\n\n";

  cout << "El primero y \u00a3ltimo elemento de dq como"
       << " lo indican rbegin() y rend()-1:\n"
       << *dq.rbegin() << ", " << *(dq.rend()-1) << "\n\n";

  // Declara un iterador a una deque<int>.
  deque<int>::iterator itr;
```

```
  // Ahora, declara un iterador inverso a una deque<int>
  deque<int>::reverse_iterator ritr;

  // Recorre en ciclo dq en dirección directa usando un iterador.
  cout << "Se aplica un bucle al vector en direcci\u00a2n directa:\n";
  for(itr = dq.begin(); itr != dq.end(); ++itr)
    cout << *itr << " ";
  cout << "\n\n";
  cout << "Ahora, se usa un iterador inverso para aplicar un bucle"
       << " en direcci\u00a2n inversa:\n";

  // Recorre dq en ciclo en dirección inversa uilizando un iterador inverso.
  for(ritr = dq.rbegin(); ritr != dq.rend(); ++ritr)
    cout << *ritr << " ";
  cout << "\n\n";

  // Crea otra deque que contiene un subrango de dq.
  deque<int> dq2(dq.begin()+2, dq.end()-4);

  // Despliega el contenido de dq2 empleando un iterador.
  show("dq2 contiene un subrango de dq: ", dq2);
  cout << endl;

  // Cambia los valores de algunos de los elementos de dq2.
  dq2[1] = 100;
  dq2[2] = 88;
  dq2[4] = 99;
  show("Tras las asignaciones, dq2 ahora contiene: ", dq2);
  cout << endl;

  // Crea una deque vacía y luego le asigna una
  // secuencia que es la inversa de  dq.
  deque<int> dq3;
  dq3.assign(dq.rbegin(), dq.rend());
  show("dq3 contiene la inversa de dq: ", dq3);
  cout << endl;

  // Incluye un elemento al frente de dq.
  dq.push_front(-31416);
  show("dq tras usar push_front(): ", dq);
  cout <<endl;

  // Ahora, limpia dq al eliminar elementos de uno en uno.
  cout << "Al eliminar elementos al frente de dq.\n";
  while(dq.size() > 0) {
    cout << "Eliminando: " << dq.front() << endl;
    dq.pop_front();
  }
  if(dq.empty()) cout << "Ahora dq est\u00a0 vac\u00a1a.\n";

   return 0;
}

// Despliega el contenido de una deque<int>.
void show(const char *msg, deque<int> q) {
```

```
  cout << msg;
  for(unsigned i=0; i < q.size(); ++i)
    cout << q[i] << " ";
  cout << "\n";
}
```

Aquí se muestra la salida:

```
El contenido de dq: 0 1 4 9 16 25 36 49 64 81
El promedio de los elementos es 28

dq tras incluir elementos al final: 0 1 4 9 16 25 36 49 64 81 100 121

dq tras usar back-pop con un elemento: 0 1 4 9 16 25 36 49 64 81 100

El primero y último elemento de dq como lo indican begin() y end()-1:
0, 100

El primero y último elemento de dq como lo indican rbegin() y rend()-1:
100, 0

Se aplica un bucle al vector en dirección directa:
0 1 4 9 16 25 36 49 64 81 100

Ahora, se usa un iterador inverso para aplicar un bucle en dirección inversa:
100 81 64 49 36 25 16 9 4 1 0

dq2 contiene un subrango de dq: 4 9 16 25 36

Tras las asignaciones, dq2 ahora contiene: 4 100 88 25 99

dq3 contiene la inversa de dq: 100 81 64 49 36 25 16 9 4 1 0

dq tras usar push_front(): -31416 0 1 4 9 16 25 36 49 64 81 100

Al eliminar elementos al frente de dq.
Eliminando: -31416
Eliminando: 0
Eliminando: 1
Eliminando: 4
Eliminando: 9
Eliminando: 16
Eliminando: 25
Eliminando: 36
Eliminando: 49
Eliminando: 64
Eliminando: 81
Eliminando: 100
dq está vacía.
```

Opciones

Aunque las funciones **push_front()** y **pop_front()** le permiten usar **deque** como una pila tipo primero en entrar último en salir, la STL ofrece un método mejor. El adaptador de contenedor **stack** proporciona una implementación que aplica este tipo de pila y provee las funciones clásicas **push()** y **pop()**. En el mismo sentido, aunque podría usar **deque** para crear una cola primero en entrar primero en salir al emplear **push_front()** y **pop_back()**, el adaptador de contenedor **queue** es una mejor posibilidad. Como opción predeterminada, tanto **stack** como **queue** usan un contenedor **deque** para contener los elementos. (Consulte *Use los adaptadores de contenedor de secuencias: **stack**, **queue** y **priority_queue***.)

Como **vector**, **deque** también ofrece otra forma de **assign()** que le permite asignar un valor a **deque**. Se muestra aquí:

void assign(size_type *num*, const T& *val*)

Esta versión elimina cualquier elemento previamente incluido en el contenedor y luego le asigna *num* copias de *val*. Podría usar esta versión de **assign()** para reinicializar una cola de dos extremos para un valor conocido, por ejemplo.

Como **vector**, **deque** *no* almacena elementos en orden. Sin embargo, es posible ordenar una cola de dos extremos al usar el algoritmo **sort()**. Consulte *ordene un contenedor,* en el capítulo 4.

Como ya se explicó, **vector** y **deque** son muy similares. Para algunos usos, como cuando se necesitan pocas inserciones (sobre todo en el medio), un vector será más eficiente que una cola de dos extremos y representa una mejor elección. (Consulte *Use **vector*** para conocer más detalles.)

Use list

Componentes clave		
Encabezados	**Clases**	**Funciones**
<list>	list	void merge(list<T, Allocator> &*ob*) void push_back(const T &*val*) reverse_iterator rbegin() void remove(const T &*val*) void reverse() void sort() void splice(iterator *i*, list<T, Allocator> &*ob*) void unique()

La clase **list** implementa un contenedor de secuencias bidireccional que se establece, con frecuencia, como una lista doblemente vinculada. A diferencia de otros dos contenedores de secuencias, **vector** y **deque**, que dan soporte a acceso aleatorio, **list** sólo puede accederse de manera secuencial. Sin embargo, como las **listas** son bidireccionales, pueden accederse de adelante hacia atrás, o viceversa. La clase **list** ofrece los mismos beneficios asociados con cualquier lista doblemente vinculada: tiempos rápidos de inserción y eliminación.

Por supuesto, el acceso a un elemento específico en la lista es una operación más lenta. Una **list** es particularmente útil cuando se agregarán con frecuencia elementos a la parte media del contenedor, o se eliminarán elementos de ésta, y no es necesario el acceso directo. En esta solución se demuestran los aspectos clave de **list**.

NOTA *El eje de la solución está en los atributos y las características de* ***list*** *que la hacen única. Consulte* Técnicas básicas de contenedor de secuencias *para conocer información que aplica a todos los contenedores de secuencias.*

Paso a paso

Para usar **list** se requieren los siguientes pasos:

1. Cree una instancia de **list** del tipo deseado.
2. Agregue elementos a la lista al llamar a **insert()**, **push_front()** o **push_back()**.
3. Elimine un elemento al final de la lista al llamar a **pop_back()**. Elimine un elemento del principio de la lista al llamar a **pop_front()**.
4. Ordene una lista al llamar a **sort()**.
5. Combine dos listas ordenadas al llamar a **merge()**.
6. Una una lista a otra al llamar a **splice()**.
7. Elimine un elemento o varios elementos específicos de la lista al llamar a **remove()**.
8. Elimine elementos duplicados al llamar a **unique()**.
9. Invierta la lista al llamar a **reverse()**.

Análisis

La especificación de plantilla para **list** es:

```
template <class T, class Allocator = allocator<T> > class list
```

Aquí, **T** es el tipo de datos que se está almacenando y **Allocator** especifica el asignador, que es, como opción predeterminada, el estándar. Para usar **list**, debe incluir el encabezado **<list>**.

La clase **list** tiene los siguientes constructores:

```
explicit list(const Allocator &asign = Allocator())
explicit list(size_type num, const T &val = T (),
          const Allocator &asign = Allocator())
list(const list<T, Allocator> &ob)
template <class InIter> list(InIter inicio, InIter final,
         const Allocator &asign = Allocator())
```

La primera forma construye una lista vacía. La segunda, una lista que contiene *num* elementos con el valor *val*. La tercera, un constructor de copia de **list**. La cuarta construye una lista que contiene los elementos en el rango de *inicio* a *final*-1. El asignador usado por **list** está especificado por *asign*, que suele permitirse como opción predeterminada.

La clase **list** da soporte a iteradores bidireccionales. Por tanto, el contenedor puede accederse mediante un iterador en direcciones directa e inversa. Sin embargo, no tienen soporte las operaciones de acceso aleatorio. Por tanto, no se proporciona la función **at()** y el operador [] no está sobrecargado.

Además de las funciones de secuencia y de contenedor de secuencias reversibles requeridas, **list** implementa las siguientes opciones: **front()**, **back()**, **push_front()**, **push_back()**, **pop_front()** y **pop_back()**. Estas funciones están descritas en la revisión general y en *Técnicas básicas de contenedor de secuencias*. Análisis adicionales se encuentran en *Use vector* y *Use* ***deque***. Las únicas funciones adicionales que no se implementan son **at()** y **operator[]()**.

La clase **list** agrega varias funciones propias, incluidas **merge()**, **reverse()**, **unique()**, **remove()**, **remove_if()** y **sort()**. Estas funciones duplican la funcionalidad proporcionada por los algoritmos estándar de los mismos nombres. Están definidas por **list** porque se encuentran especialmente optimizadas para operaciones en objetos de tipo **list** y ofrecen una opción de alto rendimiento a los algoritmos estándar.

Puede agregar elementos a una lista al usar las funciones de contenedor de secuencias estándares **insert()**, **push_front** y **push_back()**. Para eliminar elementos de una lista se llama a las funciones de contenedor de secuencias estándares **erase()**, **clear()**, **pop_back()** y **pop_front()**.

La clase **list** da soporte a iteradores directos e inversos. Como los otros contenedores de secuencias, se trata de objetos de tipo **iterator** y **reverse_iterator**. Las funciones **begin()** y **end()** devuelven iteradores al principio y al final de la lista. Las funciones **rbegin()** y **rend()** devuelven iteradores inversos al final y uno antes del principio, respectivamente. Estas funciones y las técnicas necesarias para usarlas se utilizan para recorrer en ciclo un contenedor y se describen en *Técnicas básicas de contenedor de secuencias*.

El contenido de una lista no se ordena automáticamente. Sin embargo, algunas operaciones, como la mezcla, requieren una lista ordenada. Para ordenar una lista, llame a la función **sort()**. Tiene dos versiones. Aquí se muestra la usada en esta solución:

void sort()

Después de una llamada a **sort()**, la lista se ordenará de manera ascendente, con base en el orden natural de los elementos. (La segunda versión le permite especificar una función de comparación que se usará para determinar el orden de los elementos. Consulte la secuencia *Opciones* de esta solución para conocer más detalles.)

Una función particularmente poderosa implementada por **list** es **merge()**. Combina dos listas que deben ordenarse empleando el mismo criterio. Durante una mezcla, cada elemento de la lista de origen se inserta en su ubicación apropiada en la lista de destino. Por tanto, el resultado es una lista ordenada que contiene todos los elementos de las dos listas originales. La función **merge()** tiene dos versiones. Aquí se muestra la usada por esta solución:

void merge(list<T, Allocator> &*ob*)

Combina la lista ordenada pasada en *ob* con la lista que invoca ordenada. El resultado está ordenado. Después de la mezcla, la lista contenida en *ob* queda vacía.

Una operación relacionada con la mezcla es el empalme, que se realiza mediante la función **splice()**. Cuando ocurre un empalme, la lista de origen se inserta como una unidad en la lista de destino. No tiene lugar la integración elemento por elemento de las dos listas, y no es necesario que cualquiera de las listas esté ordenada. Un empalme es, en esencia, sólo una operación de cortar y pegar. Hay tres versiones de **splice()**. Aquí se muestra la usada en esta solución:

void splice(iterator *i*, list<T, Allocator> &*ob*)

El contenido de *ob* se inserta en la lista que invoca en la ubicación señalada por *i*. Después de la operación, *ob* está vacío. Un empalme tiene lugar en cualquier punto de la secuencia de destino: al frente, en medio o a la mitad. Cuando un empalme está al frente de una lista, la secuencia empalmada está insertada antes de **begin()**. Cuando ocurre un empalme al final, la secuencia empalmada se inserta antes de **end()**.

Puede eliminar un elemento específico de una lista utilizando **remove()**, que se muestra aquí:

void remove(const T &*val*)

Elimina elementos con el valor *val* de la lista que invoca. Si ningún elemento coincide con *val*, entonces la lista queda sin cambio. A primera vista, **remove()** puede parecer redundante, porque **list** también define la función **erase()**. Sin embargo, éste no es el caso. La diferencia recae en el hecho de que **erase()** requiere iteradores a los elementos que habrán de eliminarse. La función **remove()** busca automáticamente en la lista el elemento especificado.

Otra manera de eliminar elementos de una lista es mediante el uso de la función **unique()**, que elimina elementos duplicados consecutivos. Tiene dos formas. La usada en esta solución se muestra a continuación:

void unique()

Elimina elementos duplicados de la lista que invoca. Por tanto, la lista resultante no contiene elementos duplicados consecutivos. Si la lista inicial está ordenada, entonces después de aplicar **unique()**, cada elemento será único.

Para revertir una lista, utilice la función **reverse()**, que se muestra aquí:

void reverse()

Invierte todo el contenido de la lista que invoca.

La clase **list** tiene las siguientes características de rendimiento. La inserción o eliminación de elementos en una lista ocurre en tiempo constante. No importa en qué lugar de la lista acontezca la inserción o eliminación. Debido a que **list** suele implementarse como una lista vinculada, una inserción o eliminación sólo incluye la reorganización de los vínculos y no un desplazamiento de elementos o la reasignación de memoria.

A diferencia de **vector** y **deque**, la inserción en una lista no invalida iteradores o referencias a elementos. Una eliminación sólo invalida los iteradores o las referencias a los elementos eliminados. El hecho de que estas operaciones no afecten la validez de iteradores o referencias a elementos existentes hace que la clase **list** sea especialmente útil para las aplicaciones en que se desean iteradores o referencias no volátiles.

Ejemplo

Se demuestra **list**:

```
// Demuestra list

#include <iostream>
#include <list>
```

```
using namespace std;

void mostrar(const char *msj, list<char> lista);

int main() {

  // Declara dos listas.
  list<char> listaA;
  list<char> listaB;

  // Usa push_back() para dar algunos elementos a la lista.
  listaA.push_back('A');
  listaA.push_back('F');
  listaA.push_back('B');
  listaA.push_back('R');

  listaB.push_back('X');
  listaB.push_back('A');
  listaB.push_back('F');

  mostrar("El contenido original de listaA: ", listaA);
  mostrar("El contenido original de  listaB: ", listaB);
  cout << "El tama\u00a4o de listaA es " << listaA.size() << endl;
  cout << "El tama\u00a4o de listaB es "<< listaB.size() << endl;
  cout << endl;

  // Ordena listaA y listaB
  listaA.sort();
  listaB.sort();

  mostrar("El contenido ordenado de listaA: ", listaA);
  mostrar("El contenido ordenado de listaB: ", listaB);
  cout << endl;

  // Mezcla listaB en listaA.
  listaA.merge(listaB);
  mostrar("listaA tras la mezcla: " , listaA);
  if(listaB.empty()) cout << "listaB ahora est\u00a0 vac\u00a1a.\n";
  cout << endl;

  // Elimina duplicados de listaA.
  listaA.unique();
  mostrar("listaA tras llamar a unique(): ", listaA);
  cout << endl;

  // Da a listaB algunos elementos nuevos.
  listaB.push_back('G');
  listaB.push_back('H');
  listaB.push_back('P');

  mostrar("Nuevo contenido de listaB: ", listaB);
  cout << endl;

  // Ahora, empalma listaB en listaA.
```

```
  list<char>::iterator itr = listaA.begin();
  ++itr;
  listaA.splice(itr, listaB);
  mostrar("listaA tras el empalme: ", listaA);
  cout << endl;

  // Elimina A y H.
  listaA.remove('A');
  listaA.remove('H');
  mostrar("listaA tras eliminar A y H: ", listaA);
  cout << endl;

  return 0;
}

// Despliega el contenido de una list<char>.
void mostrar(const char *msj, list<char> lista) {
  list<char>::iterator itr;

  cout << msj;

  for(itr = lista.begin(); itr != lista.end(); ++itr)
    cout << *itr << " ";

  cout << "\n";
}
```

Aquí se muestra la salida:

```
El contenido original de listaA: A F B R
El contenido original de listaB: X A F
El tamaño de listaA es 4
El tamaño de listaB es 3

El contenido ordenado de listaA: A B F R
El contenido ordenado de listaB: A F X

listaA tras la mezcla: A A B F F R X
listaB ahora está vacía ().

listaA tras llamar a unique(): A B F R X

Nuevo contenido de listaB: G H P

listaA tras el empalme: A G H P B F R X

listaA tras eliminar A y H: G P B F R X
```

Opciones

El contenedor **list** le da control detallado sobre varias de sus operaciones porque diversas funciones le permiten especificar funciones de comparación o predicados que determinan sus salidas. A continuación se describen.

Cuando se ordena una instancia de **list**, hay una segunda forma de **sort()** que le permite especificar una función de comparación que se utilizará para determinar cuándo un elemento es mayor que otro. Esta versión se muestra a continuación:

template <class Comp> void sort(Comp *fucomp*)

Aquí, *fucomp* especifica un apuntador a una función que toma dos argumentos, que deben ser del mismo tipo que los elementos del contenedor que invoca. Para ordenar de manera ascendente, la función debe devolver true cuando el primer argumento es menos que el segundo. Sin embargo, puede especificar cualquier criterio de ordenamiento que desee. Por ejemplo, puede ordenar la lista en orden inverso al revertir la comparación. He aquí una función de comparación inversa que puede usarse para ordenar al revés las listas del programa anterior:

```
// Una función de comparación inversa.
bool compinv(char a, char b) {
  if (b < a) return true;
  else return false;
}
```

Observe que los operandos están invertidos en la operación **<**. Esto ocasiona que la función devuelva true si **b** es menor que **a**, lo que causa que la lista se ordene de manera descendente. (Por lo general, se usaría la comparación **a < b**, lo que provocaría que el resultado ordenado esté en orden ascendente.) He aquí cómo se usa esta función para ordenar a la inversa **listaA**:

```
listaA.sort(compinv)
```

Otro lugar en que puede especificar una función de comparación cuando trabaja con una **list** es con esta versión de la función **merge()**:

template <class Comp> void merge(list<T, Allocator> &*ob*, Comp *fucomp*)

En esta versión, la lista ordenada pasada en *ob* se mezcla con la lista que invoca ordenada con base en el orden especificado por la función *fucomp*. Después de la mezcla, la lista contenida en *ob* está vacía. Por lo general, la misma función de comparación usada para ordenar una lista también se usa para mezclar listas. Por supuesto, son posibles usos especiales en que no sucede así.

Como se explicó, puede eliminar un elemento específico al llamar a **remove()**. Sin embargo, también puede eliminar elementos que satisfagan una cierta condición al usar **remove_if()**, que se muestra aquí:

template <class UnPred> void remove_if(UnPred *pr*)

Esta función elimina elementos para los cuales el predicado unario *pr* es true. Si ningún elemento satisface el predicado, entonces la lista queda sin cambio. Podría usar **remove_if()** para eliminar todos los elementos de una lista que satisfagan alguna condición general. Por ejemplo, suponiendo el programa anterior, podría usar este predicado para eliminar todos los elementos que se encuentran entre A y G, inclusive:

```
bool mipred(char car) {
  if(car <= 'G' && car >= 'A') return true;
  return false;
}
```

Por tanto, para eliminar todas las letras de la A a la G de **listaA**, usaría esta llamada a **remove_if()**:

```
listaA.remove_if(mipred);
```

La versión de **unique()** usada por la solución elimina elementos duplicados adyacentes. Hay una segunda forma que le permite especificar un predicado binario que define lo que constituye un elemento duplicado. (En otras palabras, el predicado determina cuando dos elementos son iguales.) Aquí se muestra esta forma de **unique()**:

template <class BinPred> void unique(BinPred *pr*)

Esta forma usa *pr* para determinar cuando un elemento es igual que otro. Esto significa que usted podría usar un criterio diferente de la igualdad consciente de bits. Por ejemplo, si una lista está almacenando información de nombres y contactos, entonces podría especificar que dos elementos son iguales si sus direcciones de correo electrónico coinciden. Como opción, podría especificar un predicado que normaliza cada elemento antes de la comparación. Por ejemplo, suponiendo el programa anterior, el siguiente predicado devolverá true si dos elementos son la misma letra, independientemente de la diferencias entre mayúsculas y minúsculas. Por tanto, dada la secuencia XxABcdEe, eliminará X y E, porque están duplicadas.

```
bool ign_mayus_pred(char a, char b) {
  if(tolower(a) == tolower(b) return true;
  else return false;
}
```

Para usar **ign_mayus_pred()** llame a **unique()**, como se muestra aquí:

```
listaA.unique(ign_mayus_pred);
```

Como ya se mencionó, **list** da soporte a iteradores bidireccionales. Esto significa que una lista puede recorrerse en dirección directa o inversa. Por tanto, suponiendo el ejemplo anterior, el siguiente fragmento usa un **reverse_iterator** para desplegar el contenido de **listaA** de atrás hacia adelante:

```
list<char>::reverse_iterator ritr;
for(ritr = listaA.rbegin(); ritr != listaA.rend(); ++ritr)
  cout << *ritr << " ";
```

Use los adaptadores de contenedor de secuencias: stack, queue y priority_queue

Componentes clave		
Encabezados	**Clases**	**Funciones**
<stack>	stack	bool empty() const void pop() void push(const value_type &*val*) size_type size() const value_type &top()
<queue>	queue	value_type &back() bool empty() const value_type &front() void pop() void push(const value_type &*val*) size_type size() const
<queue>	priority_queue	bool empty() const void pop() void push(const value_type &*val*) size_type size() const const value_type &top() const

La STL proporciona tres adaptadores de contenedor, llamados **stack**, **queue** y **priority_queue**. Utilizan uno de los contenedores de secuencias como contenedor básico, adaptándolo a sus fines especiales. En esencia, un adaptador de contenedor es simplemente una interfaz muy controlada con otro contenedor. Aunque los adaptadores de contenedor están integrados en uno de los contenedores de secuencias, también son, en sí mismos, contenedores y se usan de manera muy parecida a los otros contenedores. Sólo que el acceso a sus elementos está restringido. En esta solución se demuestra su uso.

Antes de empezar, necesita destacarse un punto importante. Los adaptadores de contenedor *no* dan soporte a toda la funcionalidad de sus contenedores. Las manipulaciones permitidas por un adaptador son un subconjunto muy restringido de lo permitido por el contenedor de base. Mientras que las restricciones precisas difieren de un adaptador a otro, hay una diferencia compartida entre todos: no dan soporte a iteradores. Si los adaptadores se lo dieran, entonces sería una tarea trivial evadir la estructura de datos definida por el adaptador (como una pila) y acceder a sus elementos fuera de orden.

Paso a paso

Para usar los adaptadores de contenedor de secuencia se requieren estos pasos:

1. Cree una instancia del adaptador de contenedor, seleccionando el adecuado para su aplicación.

2. Utilice las funciones definidas por el adaptador para insertar, acceder y eliminar elementos del contenedor. Cada adaptador define su propio conjunto de estas funciones. Por ejemplo, para incluir un elemento en una **stack**, llame a **push()**. Para obtener el siguiente elemento de una **queue**, llame a **front()**.

Análisis

La clase **stack** da soporte a una pila del tipo último en entrar primero en salir. A continuación se muestra la especificación de su plantilla:

```
template <class T, class Container = deque<T> > class stack
```

Aquí, **T** es el tipo de datos que se almacenan y **Container** es el tipo de contenedor usado para contener la pila, que es **deque**, como opción predeterminada.

El adaptador **stack** tiene el siguiente constructor:

explicit **stack**(const Container &*cnt* = Container())

El constructor **stack()** crea una pila vacía. Para usar una pila, incluya el encabezado **<stack>**. El contenedor se mantiene en un objeto protegido llamado **c** de tipo **Container**.

En general, **stack** puede adaptar cualquier contenedor que dé soporte a las siguientes operaciones:

```
back()
pop_back()
push_back()
```

Por tanto, también puede usar una **list** o un **vector** como contenedor de una pila.

La clase **stack** define la función mostrada aquí. Observe que sólo puede accederse a los elementos de una pila en el orden último en entrar primero en salir. Esto impone su naturaleza de tipo pila.

Miembro	Descripción
bool empty() const	Devuelve true si la pila que invoca está vacía y false, de otra manera.
void pop()	Elimina la parte superior de la pila.
void push(const value_type &*val*)	Incluye un elemento en la pila.
size_type size() const	Devuelve el número de elementos que se encuentra en la pila.
value_type &top() const value_type &top() const	Devuelve una referencia a la parte superior de la pila.

La clase **queue** da soporte a una cola normal tipo primero en entrar primero en salir. Los elementos se insertan en una cola en un extremo y se eliminan en el otro. No es posible acceder a los elementos de ninguna otra manera. Aquí se muestran las especificaciones de la plantilla de **queue**:

```
template <class T, class Container = deque<T> > class queue
```

Aquí, **T** es el tipo de datos que se está almacenando y **Container** es el tipo de contenedor usado para contener la cola, que es **deque**, como opción predeterminada. El contenedor se mantiene en un objeto protegido llamado **c** de tipo **Container**.

El adaptador **queue** tiene el siguiente constructor:

explicit **queue**(const Container &*cnt* = Container())

El constructor **queue()** crea una cola vacía. Para usar una cola, incluya el encabezado **<queue>**.

En general, **queue** puede adaptar cualquier contenedor que dé soporte a las siguientes operaciones:

back()
front()
pop_back()
push_back()

Por tanto, también puede usar **list** como contenedor para una cola. Sin embargo, no puede usar **vector**, porque no proporciona la función **pop_front()**.

El adaptador **queue** define la función mostrada aquí. Como puede ver, restringen **queue** al proporcionar sólo acceso tipo primero en entrar primero en salir a sus elementos.

Miembro	Descripción
value_type &back() const value_type &back() const	Devuelve una referencia al último elemento de una cola.
bool empty() const	Devuelve true si la cola que invoca está vacía y false, de otra manera.
value_type &front() const value_type &front() const	Devuelve una referencia al primer elemento de la cola.
void pop()	Elimina el primer elemento de la cola.
void push(const value_type &*val*)	Agrega un elemento con el valor especificado por *val* al final de la cola.
size_type size() const	Devuelve el número de elementos que se encuentran en la cola.

La clase **priority_queue** da soporte a colas de prioridad de un solo extremo. Una cola con prioridades organiza su contenido por prioridad. A continuación se muestra la especificación de la plantilla de **priority_queue**:

template <class T, class Container = vector<T>,
class Comp = less<nombretipo Container::value_type> >
class priority_queue

Aquí **T** es el tipo de datos que se están almacenando. **Container** es el tipo de contenedor usado para contener la cola con prioridades, que es **vector**, como opción predeterminada. El contenedor se mantiene en un objeto protegido llamado **c** de tipo **Container**. **Comp** especifica el objeto de función de comparación que determina cuando un miembro tiene menor prioridad que otro. Este objeto se mantiene en un miembro protegido llamado **comp** de tipo **Compare**.

El adaptador **priority_queue** tiene los siguientes constructores:

explicit priority_queue(const Comp &*fucomp* = Comp(),
    Container &*cnt* = Container())
template <class InIter> priority_queue(InIter *inicio*, InIter *final*,
    const Comp &*fucomp* = Comp(),
    Container &*cnt* = Container())

El primer constructor **priority_queue()** crea una cola con prioridades vacía. El segundo crea una que contiene los elementos especificados por el rango *inicio* a *final*-1. Para usar **priority_queue**, incluye el encabezado **<queue>**.

En general **priority_queue** puede adaptar cualquier contenedor que soporte las siguientes operaciones:

front()
pop_back()
push_back()

El contenedor también ha de soportar iteradores de acceso aleatorio. Así, usted puede usar deque como contenedor para una cola con prioridades. Sin embargo, no puede usar **list** porque no soporta los iteradores de acceso aleatorio.

La primera clase **priority_queue** define las funciones mostradas aquí. Los elementos en una **priority_queue** sólo pueden accederse en orden de prioridad.

Miembro	Descripción
bool empty() const	Devuelve true si la cola con prioridades que invoca está vacía y false, de otra manera.
void pop()	Elimina el primer elemento de la cola con prioridades.
void push(const value_type &*val*)	Agrega un elemento a la cola con prioridades.
size_type size() const	Devuelve el número de elementos que se encuentra en la cola con prioridades.
const value_type &top() const	Devuelve una referencia al elemento con la mayor prioridad. No se elimina el elemento.

Ejemplo

En el siguiente ejemplo se muestran los tres contenedores en acción:

```
// Demuestra los adaptadores de contenedor de secuencias.

#include <iostream>
#include <string>
#include <queue>
#include <stack>

using namespace std;
```

```
int main()
{
  // Demuestra queue.
  queue<string> q;

  cout << "Demuestra una cola para cadenas.\n";

  cout << "Incluyendo uno dos tres cuatro\n";
  q.push("uno");
  q.push("dos");
  q.push("tres");
  q.push("cuatro");

  cout << "Ahora, recupera esos valores en orden primero en entrar primero en
  salir.\n";
  while(!q.empty()) {
    cout << "Recuperando ";
    cout << q.front() << "\n";
    q.pop();
  }
  cout << endl;

  // Demuestra priority_queue.
  priority_queue<int> pq;

  cout << "Demuestra priority_queue para enteros.\n";

  cout << "Recuperando 1, 3, 4, 2.\n";
  pq.push(1);
  pq.push(3);
  pq.push(4);
  pq.push(2);

  cout << "Ahora, recupera esos valores en orden de prioridad.\n";
  while(!pq.empty()) {
    cout << "Recuperando ";
    cout << pq.top() << "\n";
    pq.pop();
  }
  cout << endl;

  // Por último, demuestra stack.
  stack<char> pila;

  cout << "Demuestra stack para caracteres.\n";

  cout << "Recuperando  A, B, C y D.\n";
  pila.push('A');
  pila.push('B');
  pila.push('C');
  pila.push('D');

  cout << "Ahora, recupera esos valores en orden \u00a3ltimo en entrar primero en
  salir.\n";
```

```
  while(!pila.empty()) {
    cout << "Recuperando: ";
    cout << pila.top() << "\n";
    pila.pop();
  }

  return 0;
}
```

Aquí se muestra la salida:

```
Demuestra una cola para cadenas.
Incluyendo uno dos tres cuatro
Ahora, recupera esos valores en orden primero en entrar primero en salir.
Recuperando uno
Recuperando dos
Recuperando tres
Recuperando cuatro

Demuestra priority_queue para enteros.
Recuperando 1, 3, 4, 2.
Ahora, recupera esos valores en orden de prioridad.
Recuperando 4
Recuperando 3
Recuperando 2
Recuperando 1

Demuestra stack para caracteres.
Recuperando  A, B, C y D.
Ahora, recupera esos valores en orden último en entrar primero en salir.
Recuperando: D
Recuperando: C
Recuperando: B
Recuperando: A
```

Ejemplo adicional: use stack para crear una calculadora de cuatro funciones

Las pilas son una de las estructuras de datos más útiles en la computación. En el nivel de máquina, proporcionan el mecanismo mediante el cual puede llamarse a una subrutina. En el nivel del programa, las pilas se usan para resolver varios problemas comunes. Por ejemplo, muchas rutinas de búsqueda basadas en inteligencia artificial dependen de pilas. Además, muchos tipos de recorridos de árbol emplean una pila. Un uso interesante de una pila es una calculadora estilo sufijo. Cuando se usa este tipo de calculadora, primero se ingresan los operandos y luego la operación que desea aplicar. Por ejemplo, para sumar 10 y 12, primero ingresa el 10, después el 12 y al último +. A medida que se ingresa cada operando, se incluye en la pila. Cuando se ingresa un operador, se recuperan los dos elementos superiores, se realiza la operación y se incluye el resultado en la pila. En el siguiente programa se usa la clase **stack** para implementar esa calculadora.

```
// Una calculadora de sufijo de cuatro funciones.
#include <iostream>
#include <stack>
#include <string>
#include <cmath>
```

```
using namespace std;

int main()
{
  stack<double> pila;
  double a, b;
  string c;

  do {
    cout << ": ";
    cin >> c;
    switch(c[0]) {
      case 's': // sale de la calculadora
        break;
      case '.': // muestra la parte superior de la pila
        cout << pila.top() << "\n";
        break;
      case '+': // suma
        if(pila.size() < 2) {
          cout << "Falta el operando\n";
          break;
        }

        a = pila.top();
        pila.pop();
        b = pila.top();
        pila.pop();
        cout << a+b << "\n";
        pila.push(a+b);
        break;
      case '-': // resta
        // Ve si el usuario ingresó un número negativo.
        if(c.size() != 1) {
          // Incluye el valor en la pila.
          pila.push(atof(c.c_str()));
          break;
        }

        // De otra manera, es una resta
        if(pila.size() < 2) {
          cout << "Falta el operando\n";
          break;
        }

        a = pila.top();
        pila.pop();
        b = pila.top();
        pila.pop();
        cout << b-a << "\n";
        pila.push(b-a);
        break;
      case '*': // Multiplica
```

```
        if(pila.size() < 2) {
          cout << "Falta el operando\n";
          break;
        }

        a = pila.top();
        pila.pop();
        b = pila.top();
        pila.pop();
        cout << a*b << "\n";
        pila.push(a*b);
        break;
      case '/': // divide
        if(pila.size() < 2) {
          cout << "Falta el operando\n";
          break;
        }

        a = pila.top();
        pila.pop();
        b = pila.top();
        pila.pop();
        cout << b/a << "\n";
        pila.push(b/a);
        break;
      default:
        // Incluye el valor en la pila
        pila.push(atof(c.c_str()));
        break;
    }
  } while(c != "s");

  return 0;
}
```

Aquí se muestra una ejecución de ejemplo:

```
: 10
: 2
: /
5
: -1
: *
-5
: 2.2
: +
-2.8
: 4
: 5
: 6
: +
11
: +
15
: s
```

En su mayor parte, la operación de la calculadora es intuitiva, pero hay un par de temas que se deben tener en cuenta. En primer lugar, para ver el valor que se encuentra en la parte superior de la pila, ingrese un punto. Esto significa que necesitará anteceder valores que sean menores de 1 con un cero, como en 0.12, por ejemplo. En segundo lugar, observe que cuando una entrada empieza con un signo de menos, si es mayor que 1, se supone que el usuario está ingresando un número negativo y no está solicitando una resta.

Opciones

Siempre y cuando el contenedor cumpla con los requisitos especificados por el adaptador, cualquiera puede usarse como contenedor base. Para usar uno diferente, simplemente especifique su nombre de clase cuando se cree una instancia del adaptador. Por ejemplo, lo siguiente crea una **queue** que adapta **list** en lugar de **deque**:

```
queue<char, list<char> > q;
```

Debido a que **q** utiliza **list** en su contenedor, estará sujeto a todos los beneficios y las desventajas de **list**. Por lo general, el contenedor predeterminado es su mejor opción, pero *sí* cuenta con otra opción. Incluso podría usar su propio contenedor como base para una **queue**. El mismo principio general se aplica también a los otros adaptadores de contenedor.

Otro punto: observe que hay un espacio entre los dos paréntesis angulares de cierre que terminan la declaración anterior. Debido a una rareza en la sintaxis de C++, este espacio es necesario. Sin él, el compilador tomará por error dos paréntesis angulares de cierre como un signo de desplazamiento a la derecha (>>) y no como terminadores de plantilla anidados. Un error común es el olvido de este espacio, que puede ser difícil de encontrar porque su programa tiene un aspecto correcto.

Almacene en un contenedor objetos definidos por el usuario

Componentes clave		
Encabezados	**Clases**	**Funciones**
	Definida por el usuario	bool operator<(*tipo-usuario a*, *tipo-usuario b*) bool operator==(*tipo-usuario a*, *tipo-usuario b*)

Puede usarse un contenedor STL para almacenar objetos de clases creados por usted. Sin embargo, estas clases deben cumplir un conjunto mínimo de requisitos. En esta solución se describen éstos y se demuestra su implementación. Se crea una clase llamada **part** que encapsula el nombre y el número asociado con alguna parte, como un clavo o un tornillo. Sin embargo, puede usarse el mismo método básico para almacenar cualquier tipo de objeto dentro de cualquier tipo de contenedor.

Paso a paso

Para habilitar objetos de una clase creada por usted y que se almacene en un contenedor de secuencia se incluyen los siguientes pasos:

1. La clase debe tener un constructor de copia públicamente accesible.
2. La clase debe proporcionar un destructor públicamente accesible.
3. La clase debe proveer un operador de asignación públicamente accesible.
4. En algunos casos, la clase debe brindar un constructor predeterminado públicamente accesible.
5. En algunos casos, la clase debe proporcionar una función **operator==()** públicamente accesible.
6. En algunos casos, la clase debe dar una función **operator<()** públicamente accesible.

Para habilitar objetos de una clase creada por usted para que se almacene en un contenedor asociativo se requieren los siguientes pasos:

1. Deben cumplirse todos los requisitos descritos por un contenedor de secuencias.
2. La clase *debe* proporcionar una función **operator<()** públicamente accesible porque todos los contenedores asociativos están ordenados.

Análisis

En el caso de todos los contenedores, si un objeto habrá de almacenarse en un contenedor, entonces su clase debe proporcionar funciones públicamente accesibles:

- Constructor de copia
- Destructor
- **operator==()**

Dependiendo del uso específico, suelen necesitarse un constructor predeterminado (sin parámetros) públicamente accesible y **operator==()**. Sin embargo, un tema clave que debe comprenderse es que el constructor de copia predeterminado, el constructor sin parámetros, el destructor y el operador de asignación proporcionados automáticamente por una clase satisfacen este requisito. Por tanto, no siempre necesita declarar explícitamente estos elementos.

Con el fin de usar un contenedor de secuencias, como **vector**, con ciertos algoritmos, como **sort()**, su clase debe proporcionar una función **operator<()** que compara dos objetos. Algunos otros algoritmos, como **find()**, requieren que se proporcione una función **operator==()** y que determina cuando un objeto es igual a otro.

Para que se almacene un objeto en un contenedor asociativo, como **set** o **multiset**, debe proporcionar un **operator<()**. Los conjuntos están ordenados al usar el operador <. También es usado por las funciones **find()**, **upper_bound()**, **lower_bound()** y **equal_range()**.

Ejemplo

En el siguiente ejemplo se crea una clase llamada **parte** que encapsula el nombre y el número de una parte. Observe que están definidos **operator<()** y **operator==()**. El operador **<** permite que un contenedor que almacena objetos de **parte** habrá de operarse en algoritmos que requieren comparaciones. En el programa se demuestra esto al ordenar el vector usando el algoritmo **sort().** El operador **==** permite la igualdad de dos objetos de **parte** que se determinarán con algoritmos como **find()**, que también es usado por el programa. (Las soluciones que describen los algoritmos STL se presentan en el capítulo 4.)

```
// Almacena objetos definidos por el usuario en un vector.
//
// Los objetos que se almacenan son instancias de la
// clase parte. El operator<() y operator==() están
// definidos por objetos de parte. Esto deja que se
// apliquen varios algoritmos de operador, como
// sort() y find().
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;


// Esta clase almacena información sobre partes.
class parte {
  string nombre;
  unsigned numero;
public:
  // Constructor predeterminado.
  parte() { nombre  = ""; numero = 0; }

  // Construye un objeto completo de parte.
  parte(string n, unsigned num) {
    nombre = n;
    numero = num;
  }

  // Funciones para acceso de datos de partes.
  string obtener_nombre() { return nombre; }
  unsigned obtener_numero() { return numero; }
};

void mostrar(const char *msj, vector<parte> vect);

// Compara objetos empleando números de parte.
bool operator<(parte a, parte b)
{
  return a.obtener_numero() < b.obtener_numero();
}

// Revisa la igualdad con base en el número de parte.
bool operator==(parte a, parte b)
{
  return a.obtener_numero() == b.obtener_numero();
}

int main()
{
  vector<parte> listaparte;

  // Inicializa la lista de partes.
  listaparte.push_back(parte("tornillo", 9324));
```

```
  listaparte.push_back(parte("desarmador", 8452));
  listaparte.push_back(parte("tuerca", 6912));
  listaparte.push_back(parte("clavo", 1274));

  // Despliega el contenido del vector.
  mostrar("Lista de partes sin ordenar:\n", listaparte);
  cout << endl;

  // Usa el algoritmo sort() para ordenar la lista de partes.
  // Esto requiere que se defina operator<() para parte.
  sort(listaparte.begin(), listaparte.end());

  mostrar("Lista de partes ordenada por n\u00a3mero:\n", listaparte);

  // Usa el algoritmo find() para encontrar una parte dado su número.
  // Esto requiere que se defina operator==() para parte.
  cout << "Buscando el n\u00a3nero de parte 6912.\n";

  vector<parte>::iterator itr;
  itr = find(listaparte.begin(), listaparte.end(), parte("", 6912));
  cout << "Parte encontrada. Su nombre es " << itr->obtener_nombre() << ".\n";

  return 0;
}

// Despliega el contenido de un vector<parte>.
void mostrar(const char *msj, vector<parte> vect) {
  vector<parte>::iterator itr;

  cout << msj;
  cout << "  Parte# Nombre\n";
  for(itr=vect.begin(); itr != vect.end(); ++itr)
    cout << "  " << itr->obtener_numero() << "\t "
         << itr->obtener_nombre() << endl;;
  cout << "\n";
}
```

Aquí se muestra la salida:

```
Lista de partes sin ordenar:
  Parte# Nombre
  9324   tornillo
  8452   desarmador
  6912   tuerca
  1274   clavo


Lista de partes ordenada por número:
  Parte# Nombre
  1274   clavo
  6912   tuerca
  8452   desarmador
  9324   tornillo

Buscando el núnero de parte 6912.
Parte encontrada. Su nombre es tuerca.
```

Opciones

Un ejemplo que demuestra el almacenamiento de un objeto de clase definido por el usuario en un **set** se presenta en *Use **set** y **multiset***.

De acuerdo con la experiencia del autor, hay algunas variaciones entre compiladores en relación con la clase precisa que debe proporcionarse para objetos de esa clase que se almacenarán en el contenedor y que se operarán con algoritmos. Las necesidades descritas en esta solución están de acuerdo con las especificadas en el estándar ANSI/ISO para C++. Sin embargo, se han visto algunos casos en que deben cumplirse requisitos adicionales. Las discrepancias entre implementaciones fueron mayores en el pasado que hoy en día. No obstante, aunque en esta solución se describen requisitos generales que debe cumplir una clase para almacenarse en un contenedor, deben tratarse como directrices (en lugar de reglas inmutables) que tal vez necesite ajustar para adecuarse a su situación específica.

Técnicas básicas de contenedor asociativo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<map>	map	iterator begin() void clear() bool empty() const iterator end() size_type erase(const key_type &*c*) iterator find(const key_type &*c*) pair<iterator, bool> insert(const value_type &*val*) reverse_iterator rbegin() reverse_iterator rend() size_type size() const void swap(map<Key, T, Comp, Allocator> &*ob*)
<map>		template <class Key, class T, class Comp, class Allocator> bool operator==(const map<Key, T, Comp, Allocator> &*supizq*. const map<Key, T, Comp, Allocator> &*supder*) template <class Key, class T, class Comp, class Allocator> bool operator<(const map<Key, T, Comp, Allocator> &*supizq*, const map<Key, T, Comp, Allocator> &*supder*) template <class Key, class T, class Comp, class Allocator> bool operator>(const map<Key, T, Comp, Allocator> &*supizq*, const map<Key, T, Comp, Allocator> &*supder*)
<utility>	pair	
<utility>		template <class *TipoC*, clase *TipoV*> pair<*tipoC*, *TopiV*> make_pair(const *tipoC* &*c*, const *tipoV* &*v*)

Todos los contenedores asociativos comparten funcionalidad común, y todos se manejan, en esencia, de la misma manera. En esta solución se utiliza esta funcionalidad común para demostrar las técnicas básicas necesarias para crear y usar un contenedor asociativo.

En esta solución se muestra cómo:

- Crear un contenedor asociativo.
- Crear elementos que constan de pares clave/valor.
- Agregar elementos a un contenedor asociativo.
- Determinar el tamaño del contenedor.
- Usar un iterador para recorrer en ciclo el contenedor.
- Asignar un contenedor a otro.
- Determinar cuándo un contenedor es equivalente a otro.
- Eliminar elementos del contenedor.
- Intercambiar el contenido de un contenedor con otro.
- Determinar si un contenedor está vacío.
- Encontrar un elemento dada su clave.

La solución usa la clase **map**. En general, las técnicas descritas aquí también se aplican a los otros contenedores asociativos, como **set**, definido por la STL. Sin embargo, **map** almacena pares clave/valor en que el tipo de clave y el tipo del valor pueden diferir. El contenedor **set** almacena objetos en que la clave y el valor son parte del mismo objeto. Más aún, **map** crea un contenedor en que cada clave debe ser única. Un contenedor **multimap**, en contraste, permite claves duplicadas. Por tanto, mientras los principios generales mostrados aquí se aplican a cualquier contenedor asociativo, se necesitará cierta adaptación, dependiendo de cuál contenedor asociativo se use.

Paso a paso

Para crear y usar un contenedor asociativo se requieren estos pasos:

1. Crear una instancia del contenedor asociativo deseado. En esta solución, se usa **map**.
2. Construir objetos **pair**, que son el tipo de objetos almacenados en un **map**.
3. Agregar elementos al contenedor al llamar a **insert()**.
4. Obtener el número de elementos en el contenedor al llamar a **size()**.
5. Determinar si el contenedor está vacío (es decir, no contiene elementos) al llamar a **empty()**.
6. Eliminar elementos del contenedor al llamar a **erase()**.
7. Eliminar todos los elementos de un contenedor al llamar a **clear()**.
8. Encontrar un elemento con una clave especificada al llamar a **find()**.
9. Obtener un iterador al principio del contenedor al llamar a **begin()**. Obtener un iterador a uno después del final del contenedor al llamar a **end()**.
10. Obtener un iterador inverso al final del contenedor al llamar a **rbegin()**. Obtener un iterador a uno antes del inicio del contenedor al llamar a **rend()**.
11. Recorrer en ciclo los elementos de un contenedor mediante un iterador.

12. Intercambiar el contenido de un contenedor con otro mediante **swap()**.
13. Determinar cuando un contenedor es igual, menor que o mayor que otro.

Análisis

La STL da soporte a dos sabores básicos de contenedor asociativo: mapas y conjuntos. En un mapa cada elemento consta de un par clave/valor y el tipo de clave puede diferir del tipo del valor. En un conjunto, la clave y el valor se incrustan en el mismo objeto. Aunque mapas y conjuntos operan, en esencia, de la misma manera, un **map** se usa en la solución porque demuestra mejor las técnicas esenciales requeridas para usar cualquier contenedor asociativo.

La especificación de plantilla para **map** se muestra a continuación:

template <class Key, class T, class Comp = less<Key>
    class Allocator = allocator<T> > class map

Aquí, **Key** es el tipo de datos de las claves y **T** es el tipo de valores que se está almacenando (asignando). La función que compara dos claves está especificada por **Comp**. Observe que las opciones predeterminadas usan el objeto de función **less**. El asignador está especificado por **Allocator**, cuya opción predeterminada es el asignador estándar.

Un aspecto central de un contenedor asociativo es que mantiene una colección ordenada de elementos basados en el valor de las claves. El orden específico es determinado por la función de comparación, que es **less** como opción predeterminada. Esto significa que, como opción predeterminada, los elementos de un contenedor asociativo están almacenados en orden ascendente. Sin embargo, es posible especificar un objeto de comparación que almacena de manera diferente los elementos.

La clase **map** da soporte a tres constructores. Aquí se muestran los dos usados en esta solución:

explicit map(const Comp &*fucomp* = Comp(), const Allocator &*asign* = Allocator())
map(const map<Key, T, Componente, Allocator> &*ob*)

La primera forma construye un mapa vacío. La segunda forma construye un mapa que contiene los mismos elementos que *ob* y es el constructor de copia de **map**. El parámetro *fucomp* especifica la función de componente usada para ordenar el mapa. En casi todos los casos, puede permitir esto como opción predeterminada. El parámetro *asign* especifica el asignador, que también suele admitirse como opción predeterminada. Para usar un mapa, debe incluir el encabezado **<map>**.

El tipo de objeto contenido por un mapa es una instancia de **pair**, que es una **struct** que encapsula dos objetos. Se declara así:

```
template <class TipoC, class TipoC> struct pair {
  typedef TipoC first_type;
  typedef TipoV second_type;
  TipoC first; //para elementos de mapa, contiene la clave
  TipoV second; //para elementos de mapa, contiene el valor


  // Constructores
  pair();
  pair(const TipoC &c, const TipoV &v);
  template<class A, class B> pair(const pairA, B> &ob);

}
```

La clase **pair** puede usarse para contener cualquier par de objetos. Sin embargo, cuando se usa para contener un par clave/valor, el valor en **first** contiene la clave y el valor en **second** contiene el valor asociado con esa clave. La clase **pair** necesita el encabezado **<utility>**, que se incluye automáticamente en **<map>**.

Puede construir un **pair** al usar uno de los constructores de **pair** o empleando la función **make_pair()**, que también se declara en **<utility>**; construye un objeto **pair** basado en los tipos de datos usados como parámetros. La función **make_pair** es genérica y tiene este prototipo:

template <class *TipoC*, class *TipoV*)
  pair<*TipoC*, *TipoV*) make_pair(const *TipoC* &*c*, const *TipoV* &*v*)

Como puede ver, devuelve un objeto **pair** que consta de valores de los tipos especificados por *TipoC* y *TipoV*. La ventaja de **make_pair()** es que los tipos del objeto que se está almacenando son determinados automáticamente por el compilador en lugar de que usted los especifique en forma explícita.

Para **map**, el tipo **value_type** es un **tipedef** para **pair<const Key, T>**. Por tanto, un mapa contiene instancias de **pair**. Más aún, el tipo **iterator** definido por **map** señala a objetos de tipo **pair<Key, T>**. Por tanto, cuando una función **map** devuelve un iterador, la clave está disponible mediante el campo **first** de **pair** y el valor se obtiene mediante el campo **second** del **pair**.

Después de que se ha creado un par, los objetos de **pair** pueden agregarse a él. Una manera de hacer esto que funciona para todos los contenedores asociativos consiste en llamar a **insert()**. Todos los contenedores asociativos dan soporte por lo menos a tres versiones de **insert()**. Éste es el usado aquí:

pair<iterator, bool> insert(const value_type &*val*)

Inserta *val* en el contenedor que invoca en un punto que mantiene el orden del contenedor asociativo. (Recuerde que **value_type** es un **type_def** para **pair<const Key, T>**.) La función devuelve un objeto de **pair** que indica el resultado de la operación. Si *val* puede insertarse, el valor **bool** (que es el campo **second**) será true, y false, de otra manera. El valor **iterator** (que está en el campo **first**) señalará al elemento insertado, si se tiene éxito, o al elemento ya existente que usa la misma clave. La operación de inserción fallará si se hace un intento por insertar un elemento en un contenedor que requiere claves únicas (como **map** o **set**) y el contenedor ya incluye la clave. Un contenedor asociativo crecerá automáticamente a medida que se necesite cuando se le agreguen elementos.

Puede eliminar uno o más elementos de un contenedor asociativo al llamar a **erase()**. Tiene por lo menos tres formas. Aquí se muestra la usada en esta solución:

size_type erase(const key_type &*tc*)

Elimina del contenedor todos los elementos que tienen claves con el valor *c*. En el caso de contenedores asociativos que requieren claves únicas, una llamada a **erase()** elimina sólo un elemento. Devuelve el número de elementos eliminados, que podría ser cero o uno para un **map**.

Puede eliminar todos los elementos de un contenedor asociativo al llamar a **clear()**, que se muestra aquí:

void clear()

Puede obtener un iterador a un elemento en un contenedor asociativo que tiene una clave especificada al llamar a **find()**, que se muestra aquí:

iterator find(const key_type &*c*)

Aquí, *c* especifica la clave. Si el contenedor incluye un elemento que tiene una clave igual a *c*, **find()** devuelve un iterador al primer elemento coincidente. Si la clave no se encuentra, entonces se devuelve **end()**.

Puede determinar el número de elementos en un contenedor al llamar a **size()**. Para determinar si un contenedor está vacío, llame a **empty()**, como se muestra aquí.

```
bool empty() const
size_type size() const
```

Puede obtener un iterador al primer elemento del contenedor al llamar a **begin()**. Debido a que los contenedores asociativos están ordenados, éste siempre será el primer elemento especificado por la función de comparación. Un iterador a una parte del último elemento en la secuencia se obtiene al llamar a **end()**. Aquí se muestran estas funciones:

```
iterator begin()
iterator end()
```

Para declarar una variable que se usará como un iterador, debe especificar el tipo de iterador del contenedor. Por ejemplo, esto declara un iterador que puede señalar a elementos dentro de **map<string, int>**:

```
map<string, int>::iterator itr;
```

Puede usar iteradores para recorrer en ciclo el contenido de un contenedor asociativo. El proceso es similar al usado para recorrer en ciclo el contenido de un contenedor de secuencias. La principal diferencia es que en contenedores asociativos que almacenan pares clave/valor, el objeto señalado por el iterador es un **pair**. Por ejemplo, suponiendo un iterador declarado de manera apropiada llamado **itr**, he aquí un bucle que despliega todas las claves y los valores en un **map** llamado **mimapa**:

```
for(itr=mimapa.begin(); itr != mimapa.end() ++itr)
  cout << "Clave: " << itr->first << ", Valor:" << itr->second << endl;
```

El bucle se ejecuta hasta que **itr** es igual a **mimapa.end()**, lo que asegura, por tanto, que se desplieguen todos los elementos. Recuerde que **end()** *no* devuelve un apuntador al último elemento de un contenedor. En cambio, devuelve un apuntador a *uno después* del último elemento. Por tanto, el último elemento de un contenedor es señalado por **end()**-1.

Como se explicó en la revisión general, un contenedor reversible es aquel en que los elementos pueden recorrerse en orden inverso (de atrás hacia adelante). Todos los contenedores asociativos integrados son reversibles. Cuando se usa un contenedor reversible, puede obtener un iterador inverso al final del contenedor al llamar a **rbegin()**. Un iterador inverso a uno antes del primer elemento en el contenedor se obtiene al llamar a **rend()**. Aquí se muestran estas funciones:

```
reverse_iterator rbegin()
reverse_iterator rend()
```

También hay versiones **const** de estas funciones. Un iterador inverso se declara como un iterador regular. Por ejemplo:

```
map<string, int>::reverse_iterator ritr;
```

Puede usar un iterador inverso para recorrer en ciclo un mapa en orden inverso. Por ejemplo, dado un iterador inverso llamado **ritr**, he aquí un bucle que despliega las claves y los valores de un mapa llamado **mimapa**, de atrás al frente:

```
for(ritr=mimapa.rbegin(); ritr != mimapa.rend() ++ritr)
  cout << "Clave: " << ritr->first << ", Valor:" << ritr->second << endl;
```

El iterador inverso **ritr** empieza en el elemento señalado por **rbegin()**, que es el último elemento del contenedor. Se ejecuta hasta que es igual a **rend()**, que señala a un elemento que está uno antes del inicio del contenedor. (En ocasiones es útil pensar que *rbegin() y **rend()** devuelven apuntadores al inicio y el final de un contenedor inverso.) Cada vez que se aumenta un iterador inverso, señala al elemento anterior. Cada vez que se disminuye, señala al siguiente elemento.

El contenido de dos contenedores asociativos puede intercambiarse al llamar a **swap()**. He aquí la manera en que se declara con **map**.

void swap(map<Key, T, Comp, Allocator> &*ob*)

El contenido del contenedor que invoca se intercambia con el especificado por *ob*.

Ejemplo

En el siguiente ejemplo se usa **map** para demostrar las técnicas básicas de contenedor asociativo:

```
// Demuestra las operaciones básicas de contenedor asociativo.
//
// En este ejemplo se usa map, pero pueden aplicarse las mismas
// técnicas básicas a cualquier contenedor asociativo.

#include <iostream>
#include <string>
#include <map>

using namespace std;

void mostrar(const char *msj, map<string, int> mp);

int main() {
  // Declara un mapa vacío que contiene pares clave/valor
  // en que la clave es una cadena y el valor es un entero.
  map<string, int> m;

  // Inserta caracteres en v. Se devuelve un iterador
  // al objeto insertado.
  m.insert(pair<string, int>("Alfa", 100));
  m.insert(pair<string, int>("Gamma", 300));
  m.insert(pair<string, int>("Beta", 200));

  // Declara un iterador a un map<string, itr>.
  map<string, int>::iterator itr;

  // Despliega el primer elemento en m.
  itr = m.begin();
  cout << "El primer par clave/valor en m: "
```

```
     << itr->first << ", " << itr->second << endl;

// Despliega el último elemento en m.
itr = m.end();
--itr;
cout << "El \u00a3ltimo par clave/valor en m: "
     << itr->first << ", " << itr->second << "\n\n";

// Despliega todo el contenido de m.
mostrar("Todo el contenido de m: ", m);

// Muestra el tamaño de m, que es el número de
// elementos contenidos por m.
cout << "El tama\u00a4o de m es " << m.size() << "\n\n";

// Declara un iterador inverso a un map<string, itr>.
map<string, int>::reverse_iterator ritr;

// Ahora, muestra el contenido de m en orden inverso.
cout << "El contenido de m invertido:\n";

for(ritr=m.rbegin(); ritr != m.rend(); ++ritr)
  cout << "  " << ritr->first << ", " << ritr->second << endl;
cout << endl;

// Encuentra un elemento dada su clave.
itr = m.find("Beta");
if(itr != m.end())
  cout << itr->first << " tiene el valor " << itr->second << "\n\n";
else
  cout << "Clave no encontrada.\n\n";

// Crea otro mapa que es igual al primero.
map<string, int> m2(m);
mostrar("El contenido de m2: ", m2);

// Compara dos mapas.
if(m == m2) cout << "m y m2 son equivalentes.\n\n";

// Inserta más elementos en m y m2.
cout << "Se insertan elementos adicionales en m y m2.\n";
m.insert(make_pair("Epsilon", 99));
m2.insert(make_pair("Zeta", 88));
mostrar("El contenido de m es ahora: ", m);
mostrar("El contenido de m2 es ahora: ", m2);

// Determina la relación entre m y m2. Es una
// comparación lexicográfica. Por ello, el primer
// elemento en el contenedor determina cuál
// contenedor es menor que el otro.
if(m < m2) cout << "m es menor que m2.\n\n";

// Elimina Beta de m.
m.erase("Beta");
```

```
  mostrar("m tras eliminar Beta: ", m);
  if(m > m2) cout << "Ahora, m es mayor que m2.\n\n";

  // Intercambia el contenido de m y m2.
  cout << "Se intercambian m y m2.\n";
  m.swap(m2);
  mostrar("El contenido de m: ", m);
  mostrar("El contenido de m2: ", m2);

  // Limpia m.
  m.clear();
  if(m.empty()) cout << "m est\u00a0 vac\u00a1o.";

  return 0;
}

// Despliega el contenido de un map<string, int> al usar
// un iterador.
void mostrar(const char *msj, map<string, int> mp) {
  map<string, int>::iterator itr;

  cout << msj << endl;
  for(itr=mp.begin(); itr != mp.end(); ++itr)
    cout << "  " << itr->first << ", " << itr->second << endl;
  cout << endl;
}
```

Aquí se muestra la salida:

```
El primer par clave/valor en m: Alfa, 100
El último par clave/valor en m: Gamma, 300

Todo el contenido de m:
  Alfa, 100
  Beta, 200
  Gamma, 300

El tamaño de m es 3

El contenido de m invertido:
  Gamma, 300
  Beta, 200
  Alfa, 100

Beta tiene el valor 200

El contenido de m2:
  Alfa, 100
  Beta, 200
  Gamma, 300

m y m2 son equivalentes.
```

```
Se insertan elementos adicionales en m y m2.
El contenido de m es ahora:
  Alfa, 100
  Beta, 200
  Épsilon, 99
  Gamma, 300

El contenido de m2 es ahora:
  Alfa, 100
  Beta, 200
  Gamma, 300
  Zeta, 88

m es menor que m2.

m tras eliminar Beta:
  Alfa, 100
  Épsilon, 99
  Gamma, 300

Ahora, m es mayor que m2.

Se intercambian m y m2.
El contenido de m:
  Alfa, 100
  Beta, 200
  Gamma, 300
  Zeta, 88

El contenido de m2:
  Alfa, 100
  Épsilon, 99
  Gamma, 300

m está vacío.
```

Gran parte del programa se explica por sí solo, pero hay unos cuantos aspectos que merecen un examen de cerca. En primer lugar, observe cómo se declara un objeto de **map** mediante la línea siguiente:

```
map<string, int> m;
```

Esto declara un mapa llamado **m** que contiene pares clave/valor en que la clave es de tipo **string** y el valor es de tipo **int**. Esto significa que los tipos de objetos contenidos por **m** son casos de **pair<string, int>**. Observe que se usa la función de comparación predeterminada **less**. Esto significa que los objetos se almacenan en el mapa en orden ascendente. Además, observe que se usa el asignador predeterminado.

A continuación, los pares clave/valor se insertan en **m** al llamar a **insert()**, como se muestra aquí:

```
m.insert(pair<string, int>("Alfa", 100));
m.insert(pair<string, int>("Gamma", 300));
m.insert(pair<string, int>("Beta", 200));
```

Debido a que **m** usa la función de comparación predeterminada, el contenido se ordena automáticamente de manera ascendente con base en las claves. Por tanto, el orden de las claves en el mapa después de las llamadas anteriores a **insert()** es Alfa, Beta, Gamma, como lo confirma la salida.

A continuación, se declara un iterador al mapa mediante la línea siguiente:

```
map<string, int>::iterator itr;
```

Debido a que el tipo de iterador debe coincidir exactamente con el tipo de contenedor, es necesario especificar los mismos tipos de clave y valor. Por ejemplo, un iterador que contiene pares clave/valor de tipo **string/int** no funciona con un mapa que contiene pares clave/valor de tipo **ofstream/string**.

Luego, el programa usa el iterador para desplegar el primero y el último par clave/valor en el mapa al usar esta secuencia:

```
// Despliega el primer elemento en m.
itr = m.begin();
cout << "El primer par clave/valor en m: "
     << itr->first << ", " << itr->second << endl;

// Despliega el último elemento en m.
itr = m.end();
--itr;
cout << "El \u00a3ltimo par clave/valor en m: "
     << itr->first << ", " << itr->second << "\n\n";
```

Como se explicó, la función **rbegin()** devuelve un iterador al primer elemento en el contenedor y **end()** devuelve un iterador a uno después del último elemento. Por esto **itr** disminuye después de la llamada a **end()** para que pueda desplegarse el último elemento. Recuerde que el tipo de objeto señalado por un iterador **map** es una instancia de **pair**. La clave está contenida en el campo **first** y el valor en el **second**. Además, observe cómo los campos de **pair** se especifican al aplicar el operador **–>** a **itr** de la misma manera en que usaría **–>** con un apuntador. En general, los iteradores funcionan como apuntadores y se manejan, en esencia, de la misma manera.

A continuación, todo el contenido de **m** se despliega al llamar a **mostrar()**, que despliega el contenido de **map<string, int>** que se pasa. Preste especial atención a la manera en que se despliegan los pares clave/valor mediante el siguiente bucle **for**:

```
for(itr=mp.begin(); itr != mp.end(); ++itr)
    cout << "  " << itr->first << ", " << itr->second << endl;
```

Debido a que **end()** obtiene un iterador que señala a uno después del final del contenedor, el bucle se detiene automáticamente después de que se ha desplegado el último elemento.

Luego, el programa despliega el contenido de **m** invertido mediante el uso de un iterador inverso y un bucle que ejecuta de **m.begin()** a **m.rend()**. Como se explicó, un iterador inverso opera en el contenedor de atrás hacia adelante. Por tanto, el incremento de un iterador inverso causa que señale al elemento anterior en el contenedor.

Preste especial atención a la manera en que se comparan dos contenedores mediante el uso de los operadores **==**, **<** y **>**. En el caso de contenedores asociativos, la comparación se conduce empleando una comparación lexicográfica de los elementos, que en el caso de **map** son pares cla-

ve/valor. Aunque el término "lexicográfico" significa "orden de diccionario", su significado suele generalizarse a lo que se relaciona con STL. En el caso de comparaciones entre contenedores, dos de éstos son iguales si contienen el mismo número de elementos, en el mismo orden, y todos los elementos correspondientes son iguales. En el caso de contenedores asociativos que contienen pares clave/valor, esto significa que cada clave y valor del elemento deben coincidir. Si se encuentra una falta de coincidencia, el resultado de una comparación lexicográfica se basa en los primeros elementos que no coinciden. Por ejemplo, suponga que un mapa contiene el par:

prueba, 10

y otro contiene:

prueba, 20

Aunque las claves sean las mismas, debido a que los valores difieren, estos dos elementos no son equivalentes. Por tanto, se juzgará que el primer mapa es menor que el segundo.

Otro tema interesante es la secuencia que encuentra un elemento dada su clave. Aquí se muestra:

```
// Encuentra un elemento dada su clave.
itr = m.find("Beta");
if(itr != m.end())
  cout << itr->first << " tiene el valor " << itr->second << "\n\n";
else
  cout << "Clave no encontrada.\n\n";
```

La capacidad de encontrar un elemento dada su clave es uno de los aspectos definitorios de los contenedores asociativos. (¡Por esa razón se les denomina "contenedores asociativos"!) El método **find()** busca el contenedor que invoca una clave que coincida con una especificada como argumento. Si se encuentra, se devuelve un iterador al elemento. De otra manera, se devuelve **end()**.

Opciones

Puede contar el número de elementos en un contenedor asociativo que coincide con una clave especificada al llamar a **count()**, que se muestra aquí:

size_type count(const key_type &*c*) const

Devuelve el número de veces que ocurre *c* en el contenedor. En el caso de contenedores que requieren claves únicas, será cero o uno.

Todos los contenedores asociativos le permiten determinar un rango de elementos en que cae un elemento. Esta capacidad tiene soporte con tres funciones: **lower_bound()**, **upper_bound()** y **equal_range()**. Se muestran a continuación. (También hay versiones **const** de estas funciones.)

iterator lower_bound(const key_type &*c*)
iterator upper_bound(const key_type &*c*)
pair<iterator, iterator> equal_range(const key_type &*c*)

La función **lower_bound()** devuelve un iterador al primer elemento en el contenedor con una clave igual o mayor que *c*. La función **upper_bound()** devuelve un iterador al primer elemento en el contenedor con una clave mayor que *c*. La función **equal_range()** devuelve un par de iteradores que señalan al límite superior y el límite inferior en el contenedor en el caso de una clave específica al llamar a **equal_range()**.

Todos los contenedores asociativos dan soporte a tres formas de **insert()**. Uno se describió antes. Aquí se muestran las otras dos versiones de **insert()**:

iterator insert(iterator *i*, const value_type &*val*)
template <class InIter> void insert(InIter *inicio*, InIter *final*)

La primera forma inserta *val* en el contenedor. En el caso de contenedores asociativos que permiten duplicados, esta forma de inserción siempre tendrá éxito. De otra manera, insertará *val* sólo si su clave no está ya en el contenedor. En cualquier caso, se devuelve un iterador al elemento con la misma clave. El iterador especificado por *i* indica un buen lugar para iniciar la búsqueda del punto de inserción apropiado. Debido a que los contenedores asociativos se ordenan con base en las claves, proporcionan un buen punto de partida que puede agilizar las inserciones. La segunda forma de **insert()** inserta el rango de *inicio* a *final*-1. Las claves duplicadas se insertarán dependiendo del contenedor. En todos los casos, el contenedor asociativo resultante permanecerá ordenado con base en claves.

Además de la forma de **erase()** usada en esta solución, todos los contenedores asociativos dan soporte a otras dos formas. Aquí se muestran:

void erase(iterator *i*)
void erase(iterator *inicio*, iterator *final*)

La primera forma elimina el elemento señalado por *i*. La segunda forma elimina los elementos en el rango de *inicio* a *final*-1.

Como ya se mencionó, la STL da soporte a dos categorías de contenedores asociativos: mapas y conjuntos. Un mapa almacena pares clave/valor. Un conjunto almacena objetos en que la clave y el valor son iguales. Dentro de estas dos categorías, hay dos divisiones: los contenedores asociativos que requieren claves únicas y los que permiten claves duplicadas. Los contenedores **mapa** y **set** requieren claves únicas. Los contenedores **multimap** y **multiset** permiten claves duplicadas. Debido a que cada contenedor asociativo usa una estrategia diferente, suele bastar con elegir el mejor para una aplicación. Por ejemplo, si necesita almacenar pares clave/valor y todas las claves son únicas, use **map**. En el caso de mapas que requieren claves duplicadas, use **multimap**.

Use map

Componentes clave		
Encabezados	**Clases**	**Funciones**
<map>	map	iterator find(const key_type &*c*) pair<iterator, bool> insert(const value_type &*val*) T &operator[](const key_type &*c*)
<utility>	pair	

En esta solución se describe lo que es probablemente el contenedor de uso más amplio: **map**. Un mapa almacena pares clave/valor, y todas las claves deben ser únicas. Por tanto, dada una clave, puede encontrar fácilmente su valor. Esto hace que **map** sea especialmente útil para mantener listas de propiedades, almacenar configuraciones de atributos y opciones, o en cualquier otro lugar en que debe encontrarse un valor mediante una clave. Por ejemplo, podría usar un **map** para crear una lista de contactos que use el nombre de una persona como clave y un número de teléfono como valor. Ese mapa le permitiría recuperar fácilmente un número de teléfono a partir de un nombre. Un mapa es un contenedor ordenado, con el orden basado en las claves. Como opción predeterminada, las claves están en orden ascendente, pero es posible especificar un orden diferente.

NOTA *El mecanismo básico que se requiere para usar un contenedor asociativo, incluido **map**, se describió en* Técnicas básicas de contenedor asociativo. *La solución dada aquí se centra en los aspectos de **map** que van más allá de estas técnicas generales.*

Paso a paso

Para usar **map** se requieren estos pasos:

1. Cree una instancia de **map** del tipo deseado.
2. Agregue elementos al mapa al llamar a **insert()** o usar el operador de subíndice.
3. Obtenga o establezca el valor de un elemento al usar el operador de subíndice.
4. Encuentre un elemento específico en el mapa al llamar a **find()**.

Análisis

La clase **map** da soporte a un contenedor asociativo en que se asignan claves únicas con valores. En esencia, una clave es simplemente un nombre que se le da a un valor. Una vez que se ha almacenado un valor, puede recuperarlo al usar su clave. Por tanto, en su sentido más general, un mapa es una lista de pares clave/valor.

La especificación de plantilla para **map** se muestra a continuación:

```
template <class Key, class T, class Comp = less<key>,
            class Allocator = allocator<pair<const Key, T> > > class map
```

Aquí, **Key** es el tipo de datos de las claves. **T** es el tipo de datos de los valores almacenados, y **Comp** es una función que compara dos claves. Los siguientes constructores están definidos en **map**:

```
explicit map(const Comp &fucomp = Comp(),
          const Allocator &asign = Allocator())
map(const map<Key, T, Comp, Allocator> &ob)
template <class InIter> map<InIter inicio, InIter final,
         const Comp &fucomp = Comp(),
         const Allocator &asign = Allocator())
```

La primera forma construye un mapa vacío. La segunda, un mapa que contiene los mismos elementos que *ob* y es un constructor de copia de **map**. La tercera forma construye un mapa que con-

tiene los elementos en el rango *inicio* a *final*-1. La función especificada por *fucomp*, si está presente, determina el orden del mapa. Con más frecuencia, permitirá que *fucomp* y *asign* estén presentes, como opción predeterminada. Para usar **map**, debe incluir **<map>**.

La clase **map** da soporte a iteradores bidireccionales. Por tanto, el contenedor puede accederse mediante un iterador en direcciones directa e inversa, pero no se da soporte a las operaciones de acceso aleatorio. Sin embargo, el operador [] sí tiene soporte, pero no en su uso tradicional.

Los pares clave/valor están almacenados en un mapa como objetos de tipo **pair**. (Consulte *Técnicas básicas de contenedor asociativo* para conocer detalles sobre **pair**.) El tipo de iterador definido por **map** señala a objetos de tipo **pair<const Key, T>**. Por tanto, cuando una función **map** devuelve un iterador, la clave está disponible mediante el miembro **first** de **pair** y el valor se obtiene mediante el campo **second** de **pair**.

La clase **map** da soporte a todas las funciones estándar especificadas por contenedores asociativos, como **find()**, **count()**, **erase()**, etc. Se describen en *Técnicas básicas de contenedor asociativo*.

Pueden agregarse elementos a un mapa de dos maneras. La primera es mediante la función **insert()**. La operación general de **insert()** se describe en *Técnicas básicas de contenedor asociativo*. He aquí un resumen. Todos los contenedores asociativos dan soporte por lo menos a tres versiones de **insert()**. El usado en esta solución es:

pair<iterator, bool> insert(const value_type &*val*)

Inserta *val* en el contenedor que invoca en un punto que mantiene el orden del contenedor asociativo. En **map**, **value_type** es un **type_def** para **pair<const Key, T>**. Por tanto, esta versión de **insert()** inserta un par clave/valor en el mapa que invoca. Devuelve un objeto **pair** que indica el resultado de la operación. Como ya se explicó, **map** requiere que todas las claves sean únicas. Por tanto, si *val* contiene una clave única, la inserción tendrá éxito. En este caso, el valor **bool** del objeto **pair** devuelto (que es el campo **second**) será true. Sin embargo, si la clave especificada ya existe, entonces este valor será false. La porción **iterator** del objeto de **pair** devuelto (que es el campo **first**) señalará al objeto insertado si se tiene éxito, o a un elemento que ya existe que usa la misma clave.

La segunda manera de agregar un par clave/valor a un mapa incluye el uso de **operator[]()**. Le sorprenderá la manera en que funciona. Aquí se muestra su prototipo:

T &operator[](const key_type &*c*)

Observe que *c* (que recibe el valor de índice) no es un entero. En cambio, es un objeto que representa una clave. Esta clave se utiliza después para encontrar el valor, y la función devuelve una referencia al valor asociado con la clave. Por tanto, el operador de subíndice se implementa con **map**, para que use una clave como índice y devuelva el valor asociado con esa clave.

Para comprender mejor los efectos de **operator[]()**, resulta de ayuda trabajar con un ejemplo. Considere un mapa llamado **mapatels** que contiene pares clave/valor que constan del nombre y el número telefónico de una persona. Además, suponga que hay una entrada en el mapa que tiene la clave "Juan", con el valor "555-0001". En este caso, la siguiente instrucción despliega el número telefónico vinculado con "Juan":

```
cout << mapatels["Juan"];
```

Debido a que "555-0001" es el valor relacionado con "Juan", esta instrucción despliega 555-0001.

Hay un aspecto muy importante del operador [] que se aplica a **map** y que expande en gran medida sus capacidades. Debido a la manera en que se implementa [], *siempre se tendrá éxito*. Si la

clave que está buscando no se encuentra en el mapa, se inserta automáticamente, y su valor es el del constructor predeterminado del tipo (que es cero para los tipos integrados). Por tanto, ¡siempre encontrará cualquier clave que busque!

Como se mencionó, el valor devuelto por el operador [] es una referencia al valor asociado con la clave usada como índice. Por tanto, puede usar el operador [] en el lado izquierdo de una asignación para dar a un elemento un nuevo valor. Por ejemplo:

```
mapatels["Juan"] = "555-1234";
```

Esta instrucción asigna el número 555-1234 a la clave "Juan". Si "Juan" no se encuentra en el mapa, se agregará primero automáticamente (con un valor predeterminado para la clave) y luego se asigna el número 555-1234. Si ya existía, entonces su valor simplemente se cambia al nuevo número.

Un tema importante: si se agregan elementos al llamar a **insert()** o al usar **operator[]()**, el mapa se mantiene en orden basado en claves.

Debido a que la clase **map** da soporte a iteradores bidireccionales, puede recorrerse en direcciones directa e inversa mediante un iterador. Más aún, la clase **map** da soporte a los tipos **iterator** y **reverse_iterator**. (También se proporcionan los tipos **const** correspondientes.) Debido a que los elementos de **map** constan de objetos **pair**, los iteradores de **map** señalan a estos objetos.

Puede obtener un iterador al primer elemento en un mapa al llamar a **begin()**. Un iterador a uno después del último elemento se obtiene al llamar a **end()**. Puede obtener un iterador inverso al final del mapa al llamar a **rbegin()** y un iterador inverso al elemento que es uno antes del principio del mapa al llamar a **rend()**. Estas funciones y la técnica usada para recorrer en ciclo un contenedor asociativo mediante el uso de un iterador se describen en *Técnicas básicas de contenedor asociativo*.

Puede obtener un iterador a un elemento específico al llamar a **find()**, que se implementa como ésta para **map**:

iterator find(const key_type &*c*)

Esta función devuelve un iterador al elemento cuya clave coincide con *c*. Si no se encuentra la clave, entonces se devuelve **end()**. Una versión **const** también está disponible. Es importante comprender que, a diferencia de [], si no se encuentra la entrada que se busca, **find()** *no* creará el elemento.

La clase **map** tiene las siguientes características de rendimiento. Los mapas están diseñados por el almacenamiento eficiente de pares clave/valor. En general, la inserción o eliminación de elementos en un mapa tiene lugar en tiempo logarítmico. Hay dos excepciones. En primer lugar, un elemento que se inserta en una ubicación determinada tiene lugar en tiempo constante amortizado. Este tiempo también se consume cuando un elemento específico se elimina dado un iterador al elemento. La inserción en un mapa no invalida iteradores o referencias a elementos. Una eliminación sólo invalida iteradores o referencias a los elementos eliminados.

Ejemplo

En el siguiente ejemplo se muestra **map** en acción. Crea un contenedor que funciona como directorio telefónico, en que el nombre es la clave y el número es el valor.

```
// Demuestra map.
//
// Este programa crea una lista telefónica simple en
// que el nombre de una persona es la clave y el
```

```
// número telefónico es el valor. Por tanto, puede
// buscar un número telefónico dado un nombre.

#include <iostream>
#include <string>
#include <map>
#include <utility>

using namespace std;

void mostrar(const char *msj, map<string, string> mt);

int main() {
  map<string, string> mapatels;

  // Inserta elementos al usar operator[].
  mapatels["Juan"] = "555-1234";
  mapatels["Diana"] = "314 555-6576";
  mapatels["Carlos"] = "660 555-9843";

  mostrar("El mapa original es: ", mapatels);
  cout << endl;

  // Ahora, cambia el número telefónico de Carlos.
  mapatels["Carlos"] = "415 997-8893";
  cout << "Nuevo n\u00a3mero para Carlos: " << mapatels["Carlos"] << "\n\n";

  // Usa find() para encontrar un número.
  map<string, string>::iterator itr;
  itr = mapatels.find("Diana");
  if(itr != mapatels.end())
    cout << "El n\u00a3mero de Diana es " << itr->second << "\n\n";

  // Bucle para map en dirección inversa.
  map<string, string>::reverse_iterator ritr;
  cout << "Despliega mapatels en orden inverso:\n";
  for(ritr = mapatels.rbegin(); ritr != mapatels.rend(); ++ritr)
   cout << "  " << ritr->first << ": " << ritr->second << endl;
  cout << endl;

  // Crea un objeto pair que contendrá el resultado
  // de una llamada a insert().
  pair<map<string, string>::iterator, bool> resultado;

  // Usa insert() para agregar una entrada.
  resultado = mapatels.insert(pair<string, string>("Joel", "555-9999"));
  if(resultado.second) cout << "Joel agregado.\n";
  mostrar("mapatels tras agregar Joel: ", mapatels);

  // No se permiten claves duplicadas, como se prueba ahora.
  resultado = mapatels.insert(pair<string, string>("Joel", "555-1010"));
  if(resultado.second) cout << "Se ha agregado un duplicado de Joel added.
  error.";
  else cout << "No se permite un duplicado de Joel.\n";
```

```
  mostrar("mapatels tras tratar de agregar un duplicado a la clave Joel: ", mapatels);

  return 0;
}

// Despliega el contenido de map<string, string> al emplear
// un iterador.
void mostrar(const char *msj, map<string, string> mt) {
  map<string, string>::iterator itr;

  cout << msj << endl;

  for(itr=mt.begin(); itr != mt.end(); ++itr)
   cout << "  " << itr->first << ": " << itr->second << endl;

  cout << endl;
}
```

Aquí se muestra la salida:

```
El mapa original es:
  Carlos: 660 555-9843
  Diana: 314 555-6576
  Juan: 555-1234

Nuevo número para Carlos: 415 997-8893

El número de Diana es 314 555-6576

Despliega mapatels en orden inverso:
  Juan: 555-1234
  Diana: 314 555-6576
  Carlos: 415 997-8893

Joel agregado.
mapatels tras agregar Joel:
  Carlos: 415 997-8893
  Diana: 314 555-6576
  Joel: 555-9999
  Juan: 555-1234

No se permite un duplicado de Joel.
mapatels tras tratar de agregar un duplicado a la clave Joel:
  Carlos: 415 997-8893
  Diana: 314 555-6576
  Joel: 555-9999
  Juan: 555-1234
```

En el programa, observe cómo se usa el operador []. En primer lugar, agrega elementos a **mapatels** en las siguientes instrucciones:

```
mapatels["Juan"] = "555-1234";
mapatels["Diana"] = "314 555-6576";
mapatels["Carlos"] = "660 555-9843";
```

Cuando se crea **mapatels**, está vacía. Por tanto, cuando se ejecutan las instrucciones anteriores, no habrá elementos en **mapatels** que tengan las claves especificadas. Esto causa que la clave y el valor se agreguen. (En esencia, un objeto **pair** que contiene la clave y el valor se construye automáticamente y se agrega al mapa.)

El siguiente uso de [] cambia el número telefónico asociado con Carlos:

```
mapatels["Carlos"] = "415 997-8893";
```

Debido a que la clave "Carlos" ya está en el mapa, se encuentra su entrada, y su valor se establece es el nuevo número telefónico.

Opciones

Como se explicó, **map** contiene pares clave/valor en que cada clave es única. Si quiere usar un mapa que permita claves duplicadas, use **multimap**. Se describe en la siguiente solución.

Como se describió en *Técnicas básicas de contenedor asociativo*, todos los contenedores asociativos dan soporte a otras dos formas de **insert()** además de la usada por la solución. Una forma es especialmente útil cuando se trabaja con mapas porque le da una manera de combinar dos mapas. Se muestra aquí:

template <class InIter> void insert(InIter *inicio*, InIter *final*)

Esta función inserta el elemento en el rango de *inicio* a *final*-1 en el mapa que invoca. Los elementos se insertan de tal manera que el mapa que invoca permanece ordenado. Por supuesto, los tipos de los elementos deben coincidir con los almacenados en el mapa que invoca y duplicar elementos que no se permiten. He aquí un ejemplo de la manera en que puede usarse esta versión de **insert()**. Suponiendo el programa anterior, la siguiente secuencia crea una segunda lista telefónica llamada **amigos** y luego se agregan estos números a **mapatels**:

```
map<string, string> amigos;

amigos["Luis"] = "555-4857";
amigos["Carmen"] = "555-1101";
amigos["Laura"] = "555-0100";

// Inserta los elementos de amigos en mapatels.
mapatels.insert(amigos.begin(), amigos.end());
```

Después de que se ejecuta esta secuencia, **mapatels** contendrá todas las entradas originales, además de las contenidas en **amigos**. El **mapatels** resultante permanece en orden. El mapa **amigos** permanecerá sin cambio.

Como todos los contenedores asociativos, **map** proporciona tres formas de **erase()** que le permiten eliminar elementos de un **map**. Se describen en *Técnicas básicas de contenedor asociativo*, pero uno merece mención especial. Se muestra aquí:

size_type erase(const key_type &*c*)

La versión de **erase()** elimina el elemento con la clave pasada en *c* y devuelve el número de elementos eliminados. Sin embargo, en el caso de **map**, nunca se eliminará más de un elemento porque se duplican elementos que no se permiten. Por tanto, si la clave especificada por *c* existe en el mapa que invoca, se eliminará y se devolverá 1. De otra manera, se devolverá 0.

Use multimap

Componentes clave		
Encabezados	**Clases**	**Funciones**
<map>	multimap	size_type erase(const key_type &*c*) iterator insert(const value_type &*val*) iterator find(const key_type &*c*) iterator upper_bound(const key_type &*c*)
<utility>	pair	

Una variable de **map** es **multimap**. Como **map**, **multimap** almacena pares clave/valor. Sin embargo, en un mapa múltiple, no es necesario que las claves sean únicas. En otras palabras, una clave podría asociarse con dos o más valores diferentes. Este tipo de contenedor es útil en dos tipos generales de situaciones. En primer lugar, ayuda en casos en que no puedan evitarse claves duplicadas. Por ejemplo, un directorio telefónico en línea podría tener dos números diferentes para la misma persona. Al usar **multimap**, el nombre de una persona puede usarse como una clave que se asigna a ambos números. En segundo lugar, es muy adecuado para situaciones en que una clave describe una relación general que existe entre sus valores. Por ejemplo, los familiares podrían representarse en un mapa múltiple que usa el apellido de la familia como clave. Los valores son los nombres. Con este método, para encontrar todos los miembros de la familia Prado, simplemente usaría Prado como clave.

NOTA *Aparte de permitir claves duplicadas,* ***multimap*** *funciona de manera parecida a* ***map****, que se describe en la solución anterior. También da soporte a todas las operaciones descritas en* Técnicas básicas de contenedor asociativo. *Esta solución se concentra en los aspectos únicos de* ***multimap****.*

Paso a paso

Para usar **multimap** se requieren estos pasos:

1. Cree una instancia de **multimap** del tipo deseado.
2. Agregue elementos, que pueden incluir claves duplicadas, al mapa múltiple, al llamar a **insert()**.
3. Encuentre todos los elementos con una clave especificada al usar **find()** y **upper_bound()**.
4. Elimine todos los elementos dentro de un mapa múltiple que tenga la misma clave al usar **erase()**.

Análisis

Aquí se muestra la especificación de la plantilla **multimap**:

```
template <class Key, class T, class Comp = less<Key>,
          class Allocator = allocator<pair<const Key, T> > > class multimap
```

Aquí, **Key** es el tipo de datos de las claves, **T** es el tipo de datos de los valores que se están almacenando (incluyendo en el mapa) y **Comp** es una función que compara dos claves. Tiene los siguientes constructores:

explicit multimap(const Comp &*fucomp* = Comp(),
 const Allocator &*asign* = Allocator())
multimap(const multimap<Key, T, Comp, Allocator> &*ob*)
template <class InIter> multimap<InIter *inicio*, InIter *final*,
 const Comp &*fucomp* = Comp(),
 const Allocator &*asign* = Allocator())

La primera forma construye un mapa múltiple vacío. La segunda, un constructor de copia de **multimap**. La tercera forma construye un mapa múltiple que contiene los elementos en el rango *inicio* a *final*-1. La función especificada por *fucomp* determina el orden del mapa múltiple. El asignador usado por el mapa múltiple está especificado por *asign*. Por lo general, *fucomp* y *asign* se permiten como opción predeterminada. Para usar **multimap**, debe incluir **<map>**.

La clase **multimap** da soporte a iteradores bidireccionales. Por tanto, el contenedor puede accederse mediante un iterador en direcciones directa e inversa. A diferencia de **map**, **multimap** no da soporte al operador []. (Como no hay una asignación uno a uno de claves a valores, no es posible indizar un objeto **multimap** empleando una clave.)

En general, **multimap** se usa como **map**. La principal diferencia es que se permiten las claves duplicadas. Esta diferencia tiene su mayor impacto en dos operaciones: insertar un elemento y encontrar un elemento. Cada una se habrá de examinar, empezando con la inserción.

Puede agregar elementos a un mapa múltiple al usar la función **insert()**. Hay tres versiones de **insert()**. Aquí se muestra la usada en esta solución:

iterator insert(const value_type &*val*)

Inserta *val* (que es un objeto **pair**) en el mapa múltiple que invoca. (Al igual que **map**, **value_type** es un **typedef** para **pair<const Key, T>**.) Debido a que se permiten claves duplicadas, *val* siempre se insertará (hasta que se agote la memoria, por supuesto). La función devuelve un iterador que señala a un elemento insertado. Por tanto, **insert()** siempre tiene éxito. Esto difiere de la versión correspondiente de **insert()** usada por **map**, que falla si hay un intento de insertar un elemento duplicado.

Debido a que la característica definitoria de **multimap** es su capacidad de almacenar más de un valor para una clave determinada, esto plantea la pregunta obvia: ¿cómo encuentro todos los valores asociados con una clave? La respuesta es un poco más complicada de lo que esperaría porque la sola función **find()** es insuficiente para encontrar varias coincidencias. Recuerde que **find()** es una función que deben implementar todos los contenedores asociativos. Se define así para **multimap**:

iterator find(const key_type &*c*)

Aquí, *c* especifica la clave. Si el mapa múltiple contiene un elemento con una clave igual a *c*, **find()** devuelve un iterador al primer elemento coincidente. Si no se encuentra la clave, entonces se devuelve **end()**. (También se proporciona una versión **const** de **find()**.)

Debido a que **find()** siempre devuelve un iterador a la *primera clave coincidente*, no hay manera de hacer que pase a la siguiente. En cambio, para obtener ésta, debe incrementar el iterador devuelto por **find()**. El proceso se detiene cuando se ha encontrado la última clave coincidente.

El punto final se obtiene mediante el uso de la función **upper_bound()**. Aquí se muestra su versión que no es **const**:

iterator upper_bound(const key_type *c*)

La función **upper_bound()** devuelve un iterador al primer elemento en el contenedor con una clave mayor que *c*. En otras palabras, devuelve un iterador al elemento que viene después de los que tienen la clave que especificó. Por tanto, suponiendo algún mapa múltiple llamado **mm**, para encontrar todas las coincidencias de una clave dada, usará una secuencia como ésta:

```
itr = mm.find(clave);
if(itr != end()) {
  do
    // ...
    ++itr;
  } while(itr != mm.upperbound(clave));
}
```

En primer lugar, se hace un intento por encontrar un elemento que coincida con la clave especificada. Si se encuentra una coincidencia, entonces se ingresa en el bucle **do**. (Recuerde que **find()** devuelve **end()** si no se encuentra la clave.) Dentro del bucle, el iterador se aumenta y su valor se comprueba contra el límite superior para la clave. Este proceso continúa hasta que **itr** señala al límite superior.

Puede eliminar todos los elementos que comparten una clave dada al emplear esta forma de **erase()**:

size_type erase(const key_type &*c*)

Elimina elementos del mapa múltiple que tienen claves con el valor *c*. Devuelve el número de elementos eliminados. Se da soporte a otras dos versiones de **erase()**, que operan en iteradores.

La clase **multimap** tiene las mismas características de rendimiento que **map**. En general, la inserción o eliminación de elementos en un mapa tiene lugar en tiempo logarítmico. Las dos excepciones son cuando un elemento se inserta en una ubicación determinada y cuando un elemento específico se elimina dado un iterador a ese elemento. En estos casos, se requiere tiempo constante amortizado. La inserción en un mapa múltiple no invalida a iteradores o referencias a elementos. Una eliminación sólo invalida los iteradores o referencias a los elementos eliminados.

Ejemplo

En el siguiente ejemplo se demuestra la manera en que puede usarse **multimap** para almacenar pares clave/valor en que pueden ocurrir duplicados. Se vuelve a trabajar el programa de ejemplo usado por la solución anterior para que use un mapa múltiple en lugar de un mapa para almacenar la lista de nombres y números telefónicos.

```
// Demostración de multimap.
//
// Este programa usa un mapa múltiple para almacenar nombres
// y números telefónicos. Permite que un nombre se asocie
// con más de un número telefónico.

#include <iostream>
#include <map>
```

```
#include <string>
using namespace std;

void mostrarnums(const char *n, multimap<string, string> mp);

int main()
{
  multimap<string, string> mapatels;

  // Inserta elementos al usar operator[].
  mapatels.insert(pair<string, string>("Juan", "Casa: 555-1111"));
  mapatels.insert(pair<string, string>("Juan", "Trabajo: 555-1234"));
  mapatels.insert(pair<string, string>("Juan", "Celular: 555-2224"));

  mapatels.insert(pair<string, string>("Diana", "Casa: 314 555-6576"));
  mapatels.insert(pair<string, string>("Diana", "Celular: 314 555-8822"));

  mapatels.insert(pair<string, string>("Carlos", "Casa: 660 555-9843"));
  mapatels.insert(pair<string, string>("Carlos", "Trabajo: 660 555-1010"));
  mapatels.insert(pair<string, string>("Carlos", "Celular: 217 555-9995"));

  // Muestra todos los números telefónicos de Juan, Diana y Carlos
  mostrarnums("Juan", mapatels);
  cout << endl;
  mostrarnums("Diana", mapatels);
  cout << endl;
  mostrarnums("Carlos", mapatels);
  cout << endl;

  // Ahora elimina todos los números telefónicos de Carlos:
  cout << "Eliminando todos los n\u00a3meros de Carlos.\n";
  int cuenta = mapatels.erase("Carlos");
  cout << "Se han eliminado " << cuenta << " elementos.\n\n";

  cout << "Tras eliminar a Carlos, fallan los intentos de encontrar el
  n\u00a3mero:\n";
  mostrarnums("Carlos", mapatels);

  return 0;
}

// Muestra todos los números para un nombre dado.
void mostrarnums(const char *n, multimap<string, string> mmp) {
  multimap<string, string>::iterator itr;

  // Encuentra la primera clave coincidente.
  itr = mmp.find(n);

  // Si se encontró la clave, se despliegan todos los números
  // telefónicos que tienen esa clave.
  if(itr != mmp.end()) {
    cout << "Los n\u00a3meros de " <<  n << ": " << endl;
    do {
      cout << "  " << itr->second << endl;
```

```
      ++itr;
    } while (itr != mmp.upper_bound(n));
  }
  else
    cout << "No se han encontrado entradas para " << n << ".\n";
}
```

Aquí se muestra la salida:

```
Los números de Juan:
  Casa: 555-1111
  Trabajo: 555-1234
  Celular: 555-2224

Los números de Diana:
  Casa: 314 555-6576
  Celular: 314 555-8822

Los números de Carlos:
  Casa: 660 555-9843
  Trabajo: 660 555-1010
  Celular: 217 555-9995

Eliminando todos los números de Carlos.
Se han eliminado 3 elementos.

Tras eliminar a Carlos, fallan los intentos de encontrar el número:
No se han encontrado entradas para Carlos.
```

Hay tres características importantes en este programa. En primer lugar, observe cómo se usa **insert()** para insertar elementos con claves duplicadas en **mapatels**, que en este programa es **multimap**. Como se explicó, **insert()** siempre tendrá éxito (hasta que se agote la memoria, por supuesto) debido a que **multimap** permite claves duplicadas. En segundo lugar, tome nota de que se encuentran todos los elementos con una clave específica. Como se explicó en el análisis anterior, para encontrar todas las entradas coincidentes con una clave dada, encuentre la primera clave al llamar a **find()**. Luego, encuentre las claves coincidentes subsecuentes al incrementar el iterador devuelto por **find()** hasta que sea igual al límite superior, como se obtiene de **upper_bound()**. Por último, tome nota de que esta llamada a **erase()** elimina todos los elementos que contiene la clave "Carlos":

```
int cuenta = mapatels.erase("Carlos");
```

Si quiere eliminar un elemento específico que tenga la clave "Carlos", entonces necesitará encontrar primero la entrada que quiera borrar y eliminarla usando otra forma de **erase()**. Este procedimiento se describe en la secuencia *Opciones* de esta solución.

Opciones

Como se explicó, esta forma de **erase()** elimina todos los elementos que comparten la clave especificada:

size_type erase(const key_type &*c*)

Elimina elementos que tienen claves con el valor *c*. Si quiere eliminar uno o más elementos específicos, entonces necesitará usar una de las otras formas de **erase()**. Recuerde que todos los contenedores asociativos, incluido **multimap**, dan soporte a las siguientes formas adicionales de **erase()**:

void erase(iterator *i*)
void erase(iterator *inicio*, iterator *final*)

La primera forma elimina el elemento señalado por *i*. La segunda elimina los elementos en el rango de *inicio* a *final*-1. Puede usar estas formas para eliminar elementos específicos de un **multimap**. Por ejemplo, suponiendo el programa anterior, la siguiente secuencia elimina el número telefónico de Carlos:

```
multimap<string, string::iterator itr;

// Encuentra la primera clave coincidente.
itr = mapatels.find("Carlos");

// Ahora, busca el número telefónico específico que se eliminará.
if(itr != mapatels.end()) {
  do {
    // Si la entrada contiene el teléfono del trabajo, lo elimina.
    if(itr->second.find("Trabajo") != string::npos) {
        mapatels.erase(itr);
        break;
      }

    ++itr
  } while (itr != mapatels.upper_bound("Carlos"));
}
```

Esta secuencia funciona al encontrar el primer elemento coincidente con la clave "Carlos". Luego usa un bucle para revisar todos los elementos con la clave "Carlos" para ver si uno de ellos contiene el número telefónico del trabajo. En la lista, los números del trabajo están antecedidos por la subcadena "Trabajo", de modo que se revisa cada valor para ver si contiene la subcadena "Trabajo". Si la incluye, la entrada se elimina y se termina el bucle.

En ocasiones, es útil conocer los puntos de inicio y final de un conjunto de elementos que comparten una clave. Para realizar esto, utilice **equal_range()**, como se muestra aquí:

pair<iterator, iterator> equal_range(const key_type &*c*)

Devuelve un objeto **pair** que contiene iteradores que señalan al límite inferior (en el campo **first**) y el límite superior (en el campo **second**) en el mapa múltiple para la clave especificada. (También se proporciona una versión **const** de la función.) Aunque todos los contenedores asociativos proporcionan **equal_range()**, es más útil con los que permiten duplicar claves. Recuerde que el límite inferior es el primer elemento que tiene una clave que es igual o mayor que *c*, y el límite superior es el primer elemento que tiene una clave mayor que *c*. Suponiendo el programa anterior, he aquí un ejemplo que muestra cómo puede usarse **equal_range()** para desplegar todos los números telefónicos de Carlos:

```
multimap<string, string::iterator itr;
pair< multimap<string, string::iterator,
     multimap<string, string::iterator> pr;
pr = mapatels.equal_range("Carlos");
```

```
itr = pr.first;

cout << "Los n\u00a3meros de carlos:\n";
while(itr != pr.second) {
  cout << itr->second << ende;
  ++itr;
}
```

Use set **y** multiset

Componentes clave		
Encabezados	**Clases**	**Funciones**
<set>	set	size_type erase(const key_type &*val*) iterator find(const key_type &*val*) pair<iterator, bool> insert(const value_type &*val*)
<set>	multiset	size_type erase(const key_type &*val*) iterator find(const key_type &*val*) pair<iterator, bool> insert(const value_type &*val*) iterator upper_bound(const key_type &*val*) const

En esta solución se demuestran **set** y **multiset**. Los contenedores de **set** son similares a los de mapas, excepto que la clave y el valor no están separados entre sí. Es decir, los conjuntos almacenan objetos en que la clave es parte del valor. En realidad, si utiliza un conjunto para almacenar uno de los tipos integrados, como un entero, la clave y el valor son iguales. Los conjuntos proporcionan contenedores muy eficientes cuando no es necesario separar la clave de los datos. El contenedor **set** requiere que todas las claves sean únicas. El contenedor **multiset** permite claves duplicadas. Aparte de esta diferencia, **set** y **multiset** funcionan de maneras similares.

Debido a que **set** y **multiset** almacenan objetos en que la clave y el valor son inseparables, podría pensar inicialmente que las aplicaciones para **set** y **multiset** están muy limitadas. En realidad, cuando almacena tipos simples, como **int** o **char**, un **set** simplemente crea una lista ordenada. Sin embargo, el poder de los conjuntos se vuelve evidente cuando se almacenan los objetos. En este caso, la clave del objeto se determina con el operador < y/o == definido por la clase. Por tanto, la clave del objeto podría constar de una sola parte de éste. Esto significa que **set** puede proporcionar un medio muy eficiente para almacenar objetos que se recuperan con base en el valor de un campo definido por el objeto. Por ejemplo, podría usar **set** para almacenar objetos que contienen información de empleados, como nombre, dirección, número telefónico y un número de ID. En este caso, el número de ID podría usarse como clave. Debido a que el principal uso de **set** y **multiset** es para contener objetos en lugar de valores simples, éste es el eje de la solución.

NOTA *Las técnicas necesarias para **set** y **multiset** son similares a las usadas por **map** y **multimap**, y no se repetirán aquí sus análisis. Para conocer información general sobre el uso de contenedores asociativos, consulte* Técnicas básicas de contenedor asociativo. *Además, consulte* Use **map** *y* Use **multimap** *para conocer información relacionada.*

Paso a paso

Para usar **set** se requieren los pasos siguientes:

1. Cree una instancia de **set** del tipo deseado.
2. Agregue elementos al conjunto al llamar a **insert()**. Cada clave de elemento debe ser única.
3. Encuentre un elemento específico en un conjunto al llamar a **find()**.
4. Elimine un elemento con una clave especificada al llamar a **erase()**.

Para usar **multiset**, se requieren los pasos siguientes:

1. Cree una instancia de **multiset** del tipo deseado.
2. Agregue elementos al conjunto al llamar a **insert()**. Se permiten claves duplicadas.
3. Encuentre todos los elementos con una clave específica usando **find()** y **upper_bound()**.
4. Elimine todos los elementos que tengan la misma clave usando **erase()**.

Análisis

La clase **set** da soporte a un conjunto en que se almacenan claves únicas en orden ascendente. Aquí se muestra su especificación de plantilla:

template <class Key, class Comp, = less<Key>,
    class Allocator = allocator<Key> > class set

Aquí, **Key** es el tipo de datos de las claves (que también contienen los datos) y **Comp** es una función que compara dos claves. La clase **set** tiene los siguientes constructores:

explicit set(const Comp &*fucomp* = Comp(),
    const Allocator &*asign* = Allocator())
set(const set<Key, Comp, Allocator> &*ob*)
template <class InIter> map<InIter *inicio*, InIter *final*,
    const Comp &*fucomp* = Comp(),
    const Allocator &*asign* = Allocator())

La primera forma construye un conjunto vacío. La segunda es el constructor de copia de **set**. La tercera construye un conjunto que contiene los elementos especificados por el rango de *inicio* a *final*-1. La función especificada por *fucomp*, si está presente, determina el orden del conjunto. Como opción predeterminada, se utiliza **less**. Para usar **set** debe incluir **<set>**.

La clase **multiset** da soporte a un conjunto en que se permiten claves duplicadas. Aquí se muestra su especificación de plantilla:

```
template <class Key, class Comp, = less<Key>,
          class Allocator = allocator<Key> > class multiset
```

Aquí, **Key** es el tipo de datos de las claves y **Comp** es una función que compara dos claves. La clase **multiset** tiene los siguientes constructores:

```
explicit multiset(const Comp &fucomp = Comp(),
        const Allocator &asign = Allocator())
multiset(const multiset<Key, Comp, Allocator> &ob)
template <class InIter> map<InIter inicio, InIter final,
        const Comp &fucomp = Comp(),
        const Allocator &asign = Allocator())
```

La primera forma construye un conjunto múltiple vacío. La segunda construye un conjunto múltiple que contiene los mismos elementos que *ob*. La tercera construye un conjunto múltiple que contiene los elementos especificados por el rango de *inicio* a *final*-1. La función especificada por *fucomp*, si está presente, determina el orden del conjunto. Como opción predeterminada, **less** es la función de comparación. El encabezado para **multiset** también es **<set>**.

Tanto **set** como **multiset** dan soporte a iteradores bidireccionales. Por tanto, es posible acceder a los contenedores mediante un iterador en las direcciones directa e inversa, pero no se da soporte a las operaciones de acceso aleatorio.

Las funciones **insert()**, **erase()** y **find()** se describen en *Técnicas básicas de contenedor asociativo*. He aquí una breve revisión de las formas usadas por esta solución. Cuando se usa con **set**, esta versión de **insert()**

```
pair<iterator, bool> insert(const value_type &val)
```

fallará si *val* contiene una clave que ya se encuentra en el contenedor. (En este caso, se devuelve false en el campo **second** del objeto **pair**, y un iterador al elemento existente en el campo **first**.) Cuando se usa con **multiset**, **insert()** siempre tendrá éxito. En ambos casos, cuando **insert()** tiene éxito, el campo **first** del objeto **pair** devuelto contendrá un iterador que señala al objeto insertado.

Cuando se usa con **set**, esta forma de **erase()**

```
size_type erase(const key_type &val)
```

elimina el elemento cuya clave coincide con *val*. Cuando se usa con **multiset**, elimina todos los elementos cuyas claves coinciden con *val*. En ambos casos, se devuelve el número de elementos eliminados.

A continuación se muestra la función **find()**:

```
iterator find(const key_type &val)
```

Para **set**, devuelve un iterador al elemento cuya clave coincide con *val*. Para **multiset**, devuelve un iterador al primer elemento cuya clave coincide con *val*. Para encontrar todos los elementos con claves coincidentes, use **upper_bound()** para establecer el límite superior. Todos los elementos que se encuentran entre los señalados por **find()** y por **upper_bound()** contendrán claves coincidentes.

Como se explicó en *Almacene en un contenedor objetos definidos por el usuario*, en general, para que un objeto se almacene en un contenedor asociativo, su clase debe sobrecargar el operador <. Esto se debe a que los contenedores asociativos se ordenan al usar el operador <. El operador < también se usa con las funciones **find()**, **upper_bound()**, **lower_bound()** y **equal_range()**. Por tanto, el secreto del uso de **set** para almacenar objetos de clase es el **operator** < **()** correctamente sobrecargado. Por lo general, el operador **<** está definido de manera tal que sólo un miembro de la clase se compara. Este miembro, por tanto, forma la clave, aunque toda la clase forma el elemento. En algunos casos, también necesita definir **operator==()**.

NOTA *De acuerdo con la experiencia del autor, hay alguna variación entre compiladores precisamente en los operadores y funciones que debe definir una clase para que se almacenen instancias de esa clase en un contenedor. Esto resulta especialmente cierto en compiladores antiguos. Como resultado, tal vez encuentre que deben sobrecargarse operadores adicionales.*

Ejemplo

En el siguiente ejemplo se muestra **set** en acción. Se usa para almacenar objetos que contienen información de empleados. El ID de empleado se usa como una clave. Por tanto, **operator<()** se implementa de modo que compara ID. Observe que **operator==()** también está implementado. Este operador no es necesario para el siguiente programa, pero se necesita en algunos algoritmos, como **find()**. Por tanto, se incluye para que esté completo. (Recuerde que, dependiendo de la implementación y el uso, tal vez deban definirse otras funciones.)

```
// Demuestra set.
//
// Este ejemplo almacena objetos que contienen
// información de empleados. El ID se usa como clave.

#include <iostream>
#include <set>
#include <string>

using namespace std;

// Esta clase almacena información del empleado.
class empleado {
  string nombre;
  string ID;
  string telefono;
  string departamento;
public:
  // Constructor predeterminado.
  empleado() { ID = nombre = telefono = departamento = ""; }

  // Construye un objeto temporal usando sólo el ID, que es la clave.
  empleado(string id) { ID = id;
                        nombre = telefono = departamento = ""; }

  // Construye un objeto de empleado completo.
  empleado(string n, string id, string dept, string p)
  {
```

```
    nombre = n;
    ID = id;
    telefono = p;
    departamento = dept;
  }

  // Acceso a funciones para datos del empleado.
  string obtener_nombre() { return nombre; }
  string obtener_id() { return ID; }
  string obtener_depto() { return departamento; }
  string obtener_tel() { return telefono; }

};

// Compara objetos usando el ID.
bool operator<(empleado a, empleado b)
{
  return a.obtener_id() < b.obtener_id();
}

// Revisa la igualdad con base en ID.
bool operator==(empleado a, empleado b)
{
  return a.obtener_id() == b.obtener_id();
}

// Crea un objeto para insertar datos de empleados.
ostream &operator<<(ostream &s, empleado &o)
{
  s << o.obtener_nombre() << endl;
  s << "Emp#:  " << o.obtener_id() << endl;
  s << "Dept:  " << o.obtener_depto() << endl;
  s << "telefono: " << o.obtener_tel() << endl;

  return s;
}

int main()
{
  set<empleado> listaemps;

  // Inicializa la lista empleado.
  listaemps.insert(empleado("Sergio Prado", "9423",
                            "Atenci\u00a2n a clientes", "555-1010"));

  listaemps.insert(empleado("Susana Torres", "8723",
                            "Ventas", "555-8899"));

  listaemps.insert(empleado("Aldo Montes", "5719",
                            "Reparaciones", "555-0174"));

  // Crea un iterador al conjunto.
  set<empleado>::iterator itr = listaemps.begin();
```

```
  // Despliega el contenido del conjunto.
  cout << "El conjunto actual: \n\n";
  do {
    cout << *itr << endl;
    ++itr;
  } while(itr != listaemps.end());
  cout << endl;

  // Encuentra un empleado específico.
  cout << "Buscando al empleado 8723.\n";
  itr = listaemps.find(empleado("8723"));
  if(itr != listaemps.end()) {
    cout << "Encontrado. Su informaci\u00a2n es:\n";
    cout << *itr << endl;
  }

  return 0;
}
```

Aquí se muestra la salida:

```
El conjunto actual:

Aldo Montes
Emp#:  5719
Dept:  Reparaciones
Tel: 555-0174

Susana Torres
Emp#:  8723
Dept:  Ventas
Tel: 555-8899

Sergio Prado
Emp#:  9423
Dept:  Atención a clientes
Tel: 555-1010


Buscando al empleado 8723.
Encontrado. Su información es:
Susana Torres
Emp#:  8723
Dept:  Ventas
Tel: 555-8899
```

Ejemplo adicional: use multiset para almacenar objetos con claves duplicadas

Como ya se explicó, la diferencia entre **set** y **multiset** es que un conjunto debe contener claves únicas, pero un conjunto múltiple puede almacenar claves duplicadas. En general, **multiset** se maneja de la misma manera que **multimap**. Por ejemplo, para encontrar todos los elementos con una clave dada, llame a **find()** para obtener un iterador a la primera clave coincidente. Luego aumente

ese iterador para obtener el siguiente elemento hasta que el iterador sea igual al límite superior. (Consulte *Use **multimap*** para conocer una descripción detallada de esta técnica.) Un mecanismo similar se utiliza para encontrar un elemento específico. Encuentre la primera clave coincidente. Luego busque el elemento específico dentro del rango delimitado.

Con el siguiente programa se demuestra la manera en que un conjunto múltiple puede almacenar elementos con claves duplicadas. Se ha vuelto a trabajar el ejemplo anterior para que la clave sea el departamento en lugar de ID. Esto significa que **operator<()** ha cambiado para comparar nombres de departamento en lugar de ID. Luego el programa despliega todos los empleados en el departamento de Reparaciones. Termina al mostrar la información para Cecilia Lona en ese departamento.

```
// Demuestra multiset.
//
// Este ejemplo almacena objetos que contienen
// información de empleados. Se usa como clave
// el nombre del departamento.

#include <iostream>
#include <set>
#include <string>

using namespace std;

// Esta clase almacena información del empleado.
class empleado {
  string nombre;
  string ID;
  string telefono;
  string departamento;
public:
  // Constructor predeterminado.
  empleado() { ID = nombre = telefono = departamento = ""; }

  // Construye un objeto temporal usando sólo el departamento,
  // que es la clave.
  empleado(string d) { departamento = d;
                       nombre = telefono = ID = ""; }

  // Construye un objeto de empleado completo.
  empleado(string n, string id, string dept, string p)
  {
    nombre = n;
    ID = id;
    telefono = p;
    departamento = dept;
  }

  // Acceso a funciones para datos del empleado.
  string obtener_nombre() { return nombre; }
  string obtener_id() { return ID; }
  string obtener_depto() { return departamento; }
  string obtener_tel() { return telefono; }

};
```

```
// Compara objetos usando el departamento.
bool operator<(empleado a, empleado b)
{
  return a.obtener_depto() < b.obtener_depto();
}

// Crea un objeto para insertar datos de empleados.
ostream &operator<<(ostream &s, empleado &o)
{
  s << o.obtener_nombre() << endl;
  s << "Emp#:  " << o.obtener_id() << endl;
  s << "Dept:  " << o.obtener_depto() << endl;
  s << "Tel: " << o.obtener_tel() << endl;

  return s;
}

int main()
{
  multiset<empleado> listaemps;

  // Initialize the empleado list.
  listaemps.insert(empleado("Sergio Prado", "9423",
                            "Atenci\u00a2n a clientes", "555-1010"));

  listaemps.insert(empleado("Susana Torres", "8723",
                            "Ventas", "555-8899"));

  listaemps.insert(empleado("Aldo Montes", "5719",
                            "Reparaciones", "555-0174"));

  listaemps.insert(empleado("Cecilia Lona", "0719",
                            "Reparaciones", "555-0175"));

  // Declara un iterador al conjunto múltiple.
  multiset<empleado>::iterator itr = listaemps.begin();

  // Despliega el contenido del conjunto múltiple.
  cout << "El conjunto actual: \n\n";
  do {
    cout << *itr << endl;
    ++itr;
  } while(itr != listaemps.end());
  cout << endl;

  // Encuentra a todos los empleados en el departamento Reparaciones.

  cout << "Todos los empleados del departamento de Reparaciones:\n\n";
  empleado e("Reparaciones"); // objeto temporal que contiene la clave Reparaciones.

  itr = listaemps.find(e);
  if(itr != listaemps.end()) {
    do {
```

```
      cout << *itr << endl;
      ++itr;
    } while(itr != listaemps.upper_bound(e));
  }

  // Ahora encuentra a Cecilia Lona en Reparaciones.
  cout << "Buscando a Cecilia Lona en Reparaciones:\n";
  itr = listaemps.find(e);
  if(itr != listaemps.end()) {
    do {
      if(itr->obtener_nombre() == "Cecilia Lona") {
        cout << "Encontrada:\n";
        cout << *itr << endl;
        break;
      }
      ++itr;
    } while(itr != listaemps.upper_bound(e));
  }

  return 0;
}
```

Aquí se muestra la salida:

```
El conjunto actual:

Sergio Prado
Emp#:  9423
Dept:  Atención a clientes
Tel:   555-1010

Aldo Montes
Emp#:  5719
Dept:  Reparaciones
Tel:   555-0174

Cecilia Lona
Emp#:  0719
Dept:  Reparaciones
Tel:   555-0175

Susana Torres
Emp#:  8723
Dept:  Ventas
Tel:   555-8899


Todos los empleados del departamento de Reparaciones:

Aldo Montes
Emp#:  5719
Dept:  Reparaciones
Tel:   555-0174
```

```
Cecilia Lona
Emp#:  0719
Dept:  Reparaciones
Tel:   555-0175

Buscando a Cecilia Lona en Reparaciones:
Encontrada:
Cecilia Lona
Emp#:  0719
Dept:  Reparaciones
Tel:   555-0175
```

Opciones

Como todos los certificados asociativos, **set** y **multiset** definen tres versiones de **erase()**. Una es la descrita en esta solución. Aquí se muestran las otras formas:

void erase(iterator *i*)
void erase(iterator *inicio*, iterator *final*)

La primera forma elimina el elemento señalado por *i*. La segunda elimina los elementos en el rango de *inicio* a *final*-1. Estas formas son especialmente útiles cuando quiere eliminar un elemento específico del conjunto múltiple. Como se explicó, la forma de **erase()** usada por la solución elimina todos los elementos cuyas claves coinciden con una clave especificada. Debido a que un conjunto múltiple permite que más de un elemento tenga la misma clave, si quiere eliminar un elemento específico, entonces necesitará encontrar ese elemento y eliminarlo al usar **erase(iterator).** (Consulte *Use **multimap*** para conocer un ejemplo que use este método.)

Los contenedores **set** y **multiset** también dan soporte a las tres formas estándar de **insert()**. Entre éstas se incluye la usada por la solución y las dos formas mostradas aquí:

iterator insert(iterator *i*, const value_type &*val*)
template <class InIter> void insert(InIter *inicio*, InIter *final*)

Para el caso de **multiset**, la primera forma inserta *val* en el contenedor. Para **set**, *val* se inserta si no contiene una clave duplicada. En todos los casos, se devuelve un iterador al elemento con la misma clave. El iterador especificado por *i* indica dónde iniciar la búsqueda del punto de inserción apropiado. Debido a que los conjuntos se almacenan con base en claves, debe tratar de usar un valor para *i* que sea cercano al punto de inserción. La segunda versión inserta los elementos en el rango de *inicio* a *final*-1. Por supuesto, cuando se usan con **set**, no se insertan los elementos con claves duplicadas.

Cuando se usa un **multiset**, en ocasiones es útil saber los puntos inicial y final de un rango de elementos que comparten una clave. Para realizar esto, use **equal_range()**, que se muestra aquí:

pair<iterator, iterator> equal_range(const key_type &*c*)

Devuelve un objeto **pair** que contiene iteradores que señalan al límite inferior (en el campo **first**) y superior (en el campo **second**) en el conjunto múltiple correspondiente a la clave especificada. (También se proporciona una versión **const** de la función.) Recuerde que el límite inferior es el primer elemento que tiene una clave que es igual o mayor que *c*, y el límite superior es el primer elemento que tiene una clave mayor que *c*. Por tanto, **equal_range()** devuelve iteradores al rango de elementos que comparten una clave común.

Si quiere almacenar un conjunto de bits, considere la clase **bitset**. Utiliza el encabezado **<bitset>** y crea un contenedor especializado para valores de bits. Sin embargo, la clase **bitset** no es un contenedor plenamente formado y no es parte de la STL. Sin embargo, para algunas aplicaciones, **bitset** podría ser una mejor opción que un contenedor STL completo.

Aunque **set** y **multiset** son muy útiles en algunas aplicaciones, son preferibles **map** y **multimap** por dos razones. En primer lugar, proporcionan las implementaciones prototípicas de contenedores que contienen pares clave/valor, porque la clave está separada del valor. En segundo lugar, la clave puede cambiar sin que necesite un cambio a la implementación de **operator<()** en los objetos que se están almacenando. Por supuesto, en todos los casos, debe usar el contenedor más adecuado para sus aplicaciones.

CAPÍTULO 4

Algoritmos, objetos de función y otros componentes de STL

En esencia, STL consta de contenedores, iteradores y algoritmos. De ellos, los dos primeros fueron el eje del capítulo 3. El tema central de este capítulo son los algoritmos. Debido al gran número de algoritmos, no es posible presentar una solución para cada uno. En cambio, se muestra cómo usar algoritmos para manejar diversas situaciones de programación STL. Estas soluciones también forman una muestra representativa de técnicas que pueden generalizarse a otros algoritmos. Por tanto, si no encuentra una solución que describa directamente lo que desea hacer, tal vez pueda adaptar una. Este capítulo también incluye soluciones que demuestran otras partes clave de la STL, incluidos objetos de función, adhesivos y negadores. También hay soluciones que demuestran un adaptador de función, tres adaptadores de iterador y los iteradores de flujo.

He aquí las soluciones contenidas en este capítulo:

- Ordene un contenedor
- Encuentre un elemento en un contenedor
- Use **search()** para encontrar una secuencia coincidente
- Invierta, gire y modifique el orden de una secuencia
- Recorra en ciclo un contenedor con **for_each()**
- Use **transform()** para cambiar una secuencia
- Realice operaciones con conjuntos
- Permute una secuencia
- Copie una secuencia de un contenedor a otro
- Reemplace y elimine elementos en un contenedor
- Combine dos secuencias ordenadas
- Cree y administre un heap
- Cree un algoritmo
- Use un objeto de función integrado

- Cree un objeto de función personalizado
- Use un adhesivo
- Use un negador
- Use el adaptador de apuntador a función
- Use los iteradores de flujo
- Use los adaptadores de iterador de inserción

Revisión general de los algoritmos

Los algoritmos expanden el poder y el alcance de STL al proporcionar una base común de funcionalidad que está disponible para todos los contenedores. También ofrecen soluciones listas para usarse a varias tareas de programación difíciles. Por ejemplo, hay algoritmos que buscan en una secuencia la ocurrencia de otra, que ordenan una secuencia o que aplican una transformación a una secuencia. Junto con los contenedores e iteradores, definen la esencia de STL.

¿Por qué se necesitan los algoritmos?

Los algoritmos son uno de los tres principales componentes de STL, y ofrecen funcionalidad no proporcionada por los propios contenedores. Como se ha mostrado en el capítulo anterior, la clase contenedora incluye varias funciones que dan soporte a una amplia variedad de operaciones. Este hecho plantea la siguiente pregunta: ¿Por qué se necesitan algoritmos separados? La respuesta tiene tres partes.

En primer lugar, los algoritmos permiten que dos tipos diferentes de contenedores operen al mismo tiempo. Debido a que casi todos los algoritmos operan mediante iteradores, el mismo algoritmo puede usar iteradores a diferentes tipos de contenedores. Por ejemplo, el algoritmo **merge()** puede usarse para combinar un **vector** con una lista.

En segundo lugar, los algoritmos contribuyen a la extensibilidad de STL. Debido a que un algoritmo puede operar en cualquier tipo de contenedor que reúne los requisitos mínimos, es posible crear nuevos contenedores que puedan manipularse mediante algoritmos estándar. Siempre y cuando un contenedor dé soporte a iteradores (lo que todos los contenedores deben hacer), los algoritmos de STL pueden usarlo. También es posible crear nuevos algoritmos. Siempre y cuando el nuevo opere mediante iteradores, puede aplicarse a cualquier contenedor.

En tercer lugar, los algoritmos mejoran STL. Debido a que proporcionan operaciones que pueden aplicarse a un amplio rango de contenedores, no es necesario que las funciones miembro de cada contenedor dupliquen esta funcionalidad. También le dan al programador una manera consistente de realizar una operación que puede aplicarse a cualquier tipo de contenedor.

Los algoritmos son funciones de plantilla

Los algoritmos de STL son funciones de plantilla. Esto significa que pueden aplicarse a cualquier tipo de contenedor. Con muy pocas excepciones, los algoritmos operan mediante iteradores. (Las excepciones usan parámetros de referencia.) Todos los algoritmos de STL requieren el encabezado **<algorithm>**.

En las descripciones de algoritmos encontradas en este capítulo, se usan los siguientes nombres de tipo de iterador genérico.

Nombre genérico	Representa
BiIter	Iterador bidireccional
ForIter	Iterador directo
InIter	Iterador de entrada
OutIter	Iterador de salida
RandIter	Iterador de acceso aleatorio

No todos los algoritmos funcionarán con todos los tipos de iteradores. Por ejemplo, el algoritmo **sort()** requiere iteradores de acceso aleatorio. Esto significa que **sort()** puede usarse en contenedores de **list**, por ejemplo. (Por esto es por lo que **list** proporciona sus propias funciones para ordenar listas.) Cuando se elige un algoritmo, debe asegurarse de que el contenedor en que se operará proporciona los iteradores necesarios.

Además de los iteradores, los prototipos de algoritmos a menudo especifican varios otros nombres de tipo genéricos, que se usan para representar predicados, funciones de comparación, etc. Aquí se muestran los usados en este capítulo:

T	Algún tipo de datos
Size	Algún tipo de entero
Func	Algún tipo de función
Generator	Una función que genera objetos
BinPred	Predicado binario
UnPred	Predicado unario
Comp	Función de comparación

Las categorías de algoritmos

La STL define un amplio número de algoritmos, y es común agruparlos por categoría. Hay muchas maneras de hacerlo. Una está integrada por las categorías usadas por el estándar internacional para C++, que se muestran aquí:

- Operaciones con secuencias que no se modifican
- Operaciones con secuencias que se modifican
- Operaciones de ordenamiento y relacionadas

De la tabla 4-1 a la 4-3 se muestran los algoritmos que comparan cada una de estas categorías. Las operaciones con secuencias que no se modifican no cambian los contenedores sobre los que operan. Las operaciones que se modifican sí. La categoría de ordenamiento incluye los diversos algoritmos de orden, además de los algoritmos que necesitan una secuencia ordenada o que, de una u otra manera, ordenan una secuencia.

Aunque las categorías definidas en el estándar C++ son útiles, cada una contiene una gran cantidad de algoritmos. Otra manera de organizar los algoritmos consiste en ordenarlos en grupos más pequeños, funcionales, como los mostrados en la tabla 4-4.

Algoritmo	Objetivo
adjacent_find	Busca elementos coincidentes adyacentes dentro de una secuencia y devuelve un iterador a la primera coincidencia.
count	Devuelve el número de elementos en la secuencia.
count_if	Devuelve el número de elementos en la secuencia que satisfagan algún predicado.
equal	Determina si dos rangos son iguales.
find	Busca un valor en un rango y devuelve un iterador a la primera ocurrencia del elemento.
find_end	Busca una subsecuencia en un rango. Devuelve un iterador a la última ocurrencia de la subsecuencia dentro del rango.
find_first_of	Encuentra el primer elemento dentro de una secuencia que coincide con un elemento dentro de un rango.
find_if	Busca, en un rango, un elemento para el que un predicado unario definido por el usuario devuelve true.
for_each	Aplica una función a un rango de elementos.
mismatch	Encuentra la primera falta de coincidencia entre elementos de dos secuencias. Se devuelven iteradores a los dos elementos.
search	Busca una subsecuencia dentro de una secuencia.
search_n	Busca una secuencia de un número especificado de elementos similares.

TABLA 4-1 Algoritmos de secuencias que no se modifican.

Revisión general de objetos de función

Los objetos de función son clases que definen **operator()**. A menudo, puede emplearse un objeto en lugar de un apuntador a función, como cuando se pasa un predicado a un algoritmo. Los objetos de función ofrecen más flexibilidad que los apuntadores a funciones, y pueden ser más eficientes en algunas situaciones. La STL proporciona muchos objetos de función integrados, como **less** y **minus**. También puede definir los propios.

Hay dos tipos de objetos de función: unarios y binarios. Un objeto de función unaria requiere un argumento; uno binario requiere dos. Debe usar el tipo de función requerido. Por ejemplo, si un algoritmo está esperando una función binaria, debe pasarle un objeto de función binaria.

Aquí se muestran los objetos de función binaria integrados:

plus	minus	multiplies	divides	modulus
equal_to	not_equal_to	greater	greater_equal	less
less_equal	logical_and	logical_or		

Algoritmo	Propósito
copy	Copia una secuencia.
copy_backward	Igual que **copy()**, excepto que mueve los elementos del final de la primera secuencia.
fill	Llena un rango con el valor especificado.
fill_n	Asigna un número específico de elementos con un valor especificado.
generate	Asigna los elementos en uno de los valores devueltos por una función generadora.
generate_n	Asigna a un número especificado de elementos los valores devueltos por una función generadora.
iter_swap	Intercambia los valores señalados por sus dos argumentos de iterador.
partition	Organiza una secuencia de modo tal que todos los elementos para los que un predicado devuelve true vengan antes de aquellos para los que el predicado devuelve false.
random_shuffle	Organiza aleatoriamente una secuencia.
replace	Reemplaza elementos en una secuencia.
replace_copy	Reemplaza elementos mientras se copia.
replace_copy_if	Mientras se copia, reemplaza elementos para los que un predicado unario definido por el usuario es true.
replace_if	Reemplaza elementos para los que un predicado unario definido por el usuario es true.
remove	Elimina elementos de un rango especificado.
remove_copy	Elimina y copia elementos de un rango especificado.
remove_copy_if	Mientras se copia, elimina elementos de un rango especificado para el que un predicado unario definido por el usuario es true.
remove_if	Elimina elementos de un rango especificado para el que un predicado unario definido por el usuario es true.
reverse	Invierte el orden de un rango.
reverse_copy	Invierte el orden de un rango mientras se copia.
rotate	Gira a la izquierda los elementos en un rango.
rotate_copy	Gira a la izquierda los elementos en un rango mientras se copia.
stable_partition	Organiza una secuencia de modo tal que todos los elementos para los que un predicado devuelve true vengan antes de aquellos para los que el predicado devuelve false. El particionamiento es estable. Esto significa que se preserva el ordenamiento relativo de la secuencia.
swap	Intercambia dos valores.
swap_ranges	Intercambia elementos en un rango.
transform	Aplica una función a un rango de elementos y almacena la salida en una nueva secuencia.
unique	Elimina elementos duplicados de un rango.
unique_copy	Elimina elementos duplicados de un rango mientras se copia.

TABLA 4-2 Operaciones de secuencias que se modifican.

Algoritmo	Propósito
binary_search	Realiza una búsqueda binaria en una secuencia ordenada.
equal_range	Devuelve un rango en que puede insertarse un elemento en una secuencia sin modificar el orden de la misma.
includes	Determina si una secuencia incluye todos los elementos en otra secuencia.
inplace_merge	Mezcla un rango con otro. Ambos rangos deben ordenarse en orden creciente. La secuencia resultante se ordena.
lexicographical_compare	Compara lexicográficamente una secuencia con otra.
lower_bound	Encuentra el primer punto en la secuencia que no es menor que un valor especificado.
make_heap	Construye un heap a partir de una secuencia.
max	Devuelve el máximo de dos valores.
max_element	Devuelve un iterador al elemento máximo dentro de un rango.
merge	Mezcla dos secuencias ordenadas, colocando el resultado en una tercera secuencia.
min	Devuelve el mínimo de dos valores.
min_element	Devuelve un iterador al elemento mínimo dentro de un rango.
next_permutation	Construye la siguiente permutación de una secuencia.
nth_element	Organiza una secuencia de modo tal que todos los elementos menores que un elemento *E* especificado vengan antes de ese elemento y todos los elementos mayores que *E* vengan después de él.
partial_sort	Ordena un rango.
partial_sort_copy	Ordena un rango y luego copia todos los elementos que quepan dentro de una secuencia resultante.
pop_heap	Intercambia el primero y el último-1 elementos y luego reconstruye el heap.
prev_permutation	Construye la permutación previa de una secuencia.
push_heap	Incluye un elemento al final de una heap.
set_difference	Produce una secuencia que contiene la diferencia entre dos conjuntos ordenados.
set_intersection	Produce una secuencia que contiene la intersección entre dos conjuntos ordenados.
set_symmetric_difference	Produce una secuencia que contiene la diferencia simétrica entre dos conjuntos ordenados.
set_union	Produce una secuencia que contiene la unión de dos conjuntos ordenados.
sort	Ordena un rango.
sort_heap	Ordena un heap dentro de un rango especificado.
stable_sort	Ordena un rango. El orden es estable. Esto significa que elementos iguales no se reorganizan.
upper_bound	Encuentra el último punto en una secuencia que no es mayor que algún valor.

TABLA 4-3 Algoritmos de ordenamiento y relacionados.

Copia			
copy	copy_backward	iter_swap	fill
fill_n	swap	swap_ranges	
Secuencias no ordenadas de búsqueda			
adjacent_find	equal	find	find_end
find_if	find_first_of	mismatch	search
search_n			
Elementos de reemplazo y eliminación			
remove	remove_if	remove_copy	remove_copy_if
replace	replace_if	replace_copy	replace_copy_if
unique	unique_copy		
Reordenamiento de una secuencia			
rotate	rotate_copy	random_shuffle	partition
reverse	reverse_copy	stable_partition	next_permutation
prev_permutation			
Ordenamiento y búsqueda de una secuencia ordenada			
nth_element	sort	stable_sort	partial_sort
partial_sort_copy	binary_search	lower_bound	upper_bound
equal_range			
Mezcla de secuencias ordenadas			
merge	inplace_merge		
Operaciones con conjuntos			
includes	set_difference	set_intersection	set_symmetric_difference
set_union			
Operaciones con heap			
make_heap	push_heap	pop_heap	sort_heap
Mínimo y máximo			
max	max_element	min	min_element
Transformación y generación de una secuencia			
generate	generate_n	transform	
Varios			
count	count_if	for_each	lexicographical_compare

TABLA 4-4 Los algoritmos de STL organizados por agrupamientos funcionales.

He aquí los objetos de función unaria:

logical_not	negate

Todos los objetos de función integrados son clases de plantilla que sobrecargan **operator()**. Debido a que son clases de plantilla, pueden trabajar en cualquier tipo de datos para los que están definidas las operaciones asociadas. Los objetos de función integrados usan el encabezado **<functional>**.

Aunque es permisible construir un objeto de función por anticipado, a menudo lo construirá cuando se pasa a un algoritmo. Se hace esto al llamar explícitamente a su constructor con el uso de la siguiente forma general:

func_ob<*tipo*>()

Por ejemplo,

```
sort(inicio, final, mayor<int>())
```

construye un objeto **mayor** para usarlo en operandos de tipo **int** y lo pasa al algoritmo **sort()**.

Hay un tipo especial de objeto de función llamado *predicado*. La característica definida de un predicado es que devuelve un valor **bool**. En otras palabras, un predicado devuelve un resultado true/false. Hay predicados unarios y binarios. Uno unario toma un argumento. Uno binario toma dos. Hay un tipo especial de predicado que realiza una comparación menor que, devolviendo true sólo si el elemento es menor que otro. A este tipo de predicado se le denomina en ocasiones *función de comparación*.

Revisión general de adhesivos y negadores

Como se explicó en la sección anterior, un objeto de función binaria toma dos parámetros. Por lo general, éstos reciben valores de la secuencia o las secuencias bajo las que el objeto está operando. Sin embargo, habrá ocasiones en que querrá que uno de los valores se una a un valor específico. Por ejemplo, tal vez quiera usar **less** para comparar elementos de una secuencia contra un valor especificado. Para manejar este tipo de situación, usará un *adhesivo*. La STL proporciona dos adhesivos: **bind1st()** y **bind2nd()**. El primero une un valor con el primer argumento de un objeto de función binaria. El segundo une un valor al segundo argumento.

Relacionados con los adhesivos están los *negadores*. Éstos son **not1()** y **not2()**. Devuelven la negación (es decir, el complemento) de cualquier predicado que modifiquen.

Ordene un contenedor

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template<class RandIter> void sort(RandIter *inicio*, RandIter *fin*) template<class RandIter, class Comp> void sort(RandIter *inicio*, RandIter *fin*, Comp *fucomp*)

Una de las operaciones de contenedor más comunes es el ordenamiento. La razón para esto es fácil de comprender. No se requieren contenedores de secuencia para mantener sus elementos en orden. Por ejemplo, ni **vector** ni **deque** mantienen un contenedor ordenado. Por tanto, si quiere que los elementos de uno de estos contenedores se ordenen, necesitará ordenarlo. Por fortuna, es fácil ordenar uno de estos contenedores mediante el algoritmo **sort()**. El contenedor puede ordenarse de manera natural o en un orden determinado mediante una función de comparación. Esta solución describe el proceso y ofrece tres opciones interesantes.

Paso a paso

Para ordenar un contenedor de manera natural sólo se usa un paso:

1. Llame a la forma de dos parámetros **sort()**, pasando sus iteradores al principio y el final del rango que habrá de ordenarse.

Ordenar un contenedor en un orden determinado mediante una función de comparación que proporciona incluye estos pasos:

1. Si habrá de ordenar con base en una función de comparación que se proporcionará, cree la función comparación.
2. Llame a la forma de tres parámetros de **sort()**, pasando en iteradores al principio y al final de la secuencia y a la función de comparación.

Análisis

La STL proporciona varios algoritmos de ordenamiento. En el centro, se encuentra **sort()**, que se muestra a continuación:

template <class RandIter>
 void sort(RandIter *inicio*, RandIter *final*)

template <class RandIter, class Comp>
 void sort(RandIter *inicio*, RandIter *final*, Comp *fucomp*)

El algoritmo **sort()** ordena el rango de *inicio* a *final*–1. La segunda forma le permite pasar una función de comparación a *fucomp* que determina cuando un elemento es menor que otro. Esta función puede pasarse mediante un apuntador o un objeto de función, como **greater()**. (Consulte *Use un objeto de función integrado* para conocer una solución en que se analizan los objetos de función. Consulte *Cree un objeto de función personalizado* para conocer detalles sobre la creación de su propio objeto de función.)

Observe que **sort()** requiere iteradores de acceso aleatorio. Sólo unos cuantos contenedores, como **vector** y **deque**, dan soporte a iteradores de acceso aleatorio. Estos contenedores, como **list**, que no lo dan, deben proporcionar sus propias rutinas de ordenamiento.

Es importante comprender que **sort()** ordena el rango especificado por sus argumentos, que no necesitan incluir todo el contenido del contenedor. Por tanto, **sort()** puede usarse para ordenar un subconjunto de un contenedor. Para ordenar un contenedor completo, debe especificar **begin()** y **end()** como puntos de inicio y final.

Ejemplo

En el siguiente ejemplo se muestran ambas versiones de **sort()** en acción. Se crea un vector y luego lo ordena de manera natural. Luego usa el objeto de función estándar **greater()** para ordenar el vector de manera descendente. Por último, reordena los seis elementos centrales de manera natural.

```
// Demuestra el algoritmo sort()

#include <cstdlib>
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>

using namespace std;

void mostrar(const char *msj, vector<int> vect);

int main()
{
  vector<int> v(10);

  // Inicializa v con valores aleatorios.
  for(unsigned i=0; i < v.size(); i++)
    v[i] = rand() % 100;

  mostrar("Orden original:\n", v);
  cout << endl;

  // Ordena todo el contenedor.
  sort(v.begin(), v.end());

  mostrar("Tras aplicar el orden natural:\n", v);
  cout << endl;

  // Ahora, ordena de manera descendente al usar greater().
  sort(v.begin(), v.end(), greater<int>());

  mostrar("Tras aplicar el orden descendente:\n", v);
  cout << endl;

  // Ordena un subconjunto del contenedor.
  sort(v.begin()+2, v.end()-2);
```

```
  mostrar("Tras ordenar los elementos de v[2] a v[7] de manera natural:\n", v);

  return 0;
}

// Despliega el contenedor de vector<int>.
void mostrar(const char *msj, vector<int> vect) {
  cout << msj;
  for(unsigned i=0; i < vect.size(); ++i)
    cout << vect[i] << " ";
  cout << "\n";
}
```

Aquí se muestra la salida:

```
Orden original:
41 67 34 0 69 24 78 58 62 64

Tras aplicar el orden natural:
0 24 34 41 58 62 64 67 69 78

Tras aplicar el orden descendente:
78 69 67 64 62 58 41 34 24 0

Tras ordenar los elementos de v[2] a v[7] de manera natural:
78 69 34 41 58 62 64 67 24 0
```

Opciones

Una variación interesante del ordenamiento se encuentra en **partial_sort()**. Tiene las dos versiones mostradas aquí:

template <class RandIter>
  void partial_sort(RandIter *inicio*, RandIter *medio*, RandIter *final*)

template <class RandIter, class Comp>
  void partial_sort(RandIter *inicio*, RandIter *medio*, RandIter *final*, Comp *fucomp*)

El algoritmo **partial_sort()** ordena elementos del rango *inicio* a *final*-1. Sin embargo, después de la ejecución, sólo se ordenarán los elementos en el rango *inicio* a *medio*-1. El resto está en orden arbitrario. Por tanto, **partial_sort()** examina todos los elementos de *inicio* a *final*, pero sólo ordena los elementos *medio-inicio* de todo el rango, y esos elementos son todos menos los elementos restantes, no ordenados. Podría usar **partial_sort()** para obtener las 10 canciones más vendidas de la lista de todas las canciones proporcionadas por un servicio de música en línea, por ejemplo. La segunda forma le permite especificar una función de comparación que determina cuándo un elemento es menor que otro. Suponiendo el programa de ejemplo, el siguiente fragmento ordena los primeros cinco elementos de **v**:

```
partial_sort (v.begin(), v.begin()+5, v.end));
```

Después de que se ejecuta esta instrucción, los primeros cinco elementos de **v** estarán en orden. Los elementos restantes estarán ordenados de manera no específica.

Una variación útil en el ordenamiento parcial es **partial_sort_copy()**, que pone los elementos ordenados en otra secuencia. Tiene las siguientes dos versiones:

```
template <class InIter, class RandIter>
  RandIter partial_sort_copy(InIter inicio, InIter final,
                             RandIter inicio_resultado, RandIter final_resultado)
template <class InIter, RandIter, class Comp>
  void partial_sort_copy(InIter inicio, InIter final,
                         RandIter inicio_resultado, RandIter final_resultado,
                         Comp fucomp)
```

Ambos ordenan el rango de *inicio* a *final*-1 y luego copian todos los elementos que conforman la secuencia resultante definida por *inicio_resultado* a *final_resultado*. Se devuelve un iterador a uno después del último elemento copiado en la secuencia resultante. La segunda forma le permite especificar una función de comparación que determina cuando un elemento es menor que otro.

Otra opción de ordenamiento es **stable_sort()**, que proporciona un orden que no reorganiza elementos iguales. Tiene dos formas:

```
template <class RandIter>
  void stable_sort(RandIter inicio, RandIter final)

template <class RandIter, class Comp>
  void stable_sort(RandIter inicio, RandIter final, Comp fucomp)
```

Ordena el rango de *inicio* a *final*-1, pero no se reordena los elementos iguales. La segunda forma le permite especificar una función de comparación que determina cuando un elemento es menor que otro.

Encuentre un elemento en un contenedor

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template <class InIter, class T> InIter find(InIter *inicio*, InIter *final*, const T &*val*) template <class InIter, class UnPred> InIter find_if(InIter *inicio*, InIter *final*, UnPred *funp*)

Con frecuencia, querrá encontrar un elemento específico dentro de un contenedor. Por ejemplo, tal vez quiera encontrar un elemento para que pueda eliminarse, verse o actualizarse con nueva información. Cada vez que sea necesario, la STL proporciona varios algoritmos que, de una manera u otra, le permiten encontrar un elemento específico dentro de un contenedor. En esta solución se revisan dos: **find()** y **find_if()**, pero otros varios se describen en la sección *Opciones* de esta solución. La principal ventaja de **find()** y **find_if()** es que no requieren que se ordene el contenedor. Por tanto, funcionan en todos los casos.

Paso a paso

Para usar **find()** para encontrar un elemento dentro de un contenedor se requieren estos pasos:

1. Cree una instancia de un objeto que desee encontrar.
2. Llame a **find()**, pasando en iteradores al rango de búsqueda y el objeto que se buscará.

Para usar **find_if()** para buscar un elemento dentro de un contenedor se requieren estos pasos:

1. Cree un predicado unario que devuelve true cuando se encuentra el objeto deseado.
2. Llame a **find_if()**, pasando en iteradores al rango que se ordenará y el predicado del paso 1.

Análisis

Tal vez los algoritmos de uso más extenso sean **find()** y su pariente cercano, **find_if()**. El algoritmo **find()** busca en un rango la primera aparición de un elemento especificado. Aquí se muestra:

template <class InIter, class T>
 InIter find(InIter *inicio*, InIter *final*, const T &*val*)

Busca, en el rango *inicio* a *final*-1, el valor especificado por *val*. Devuelve un iterador a la primera aparición del elemento o al *final* si *val* no está en el rango.

El algoritmo **find_if()** busca en un rango la primera aparición de un elemento que cumple las condiciones especificadas por un predicado. Aquí se muestra:

template <class InIter, class UnPred>
 InIter find_if(InIter *inicio*, InIter *final*, UnPred *funp*)

Busca en el rango de *inicio* a *final*-1 un elemento para el cual el predicado unario *funp* devuelve true. Devuelve un iterador al primer elemento que satisfaga *funp*, o al *final*, si *val* no está en el rango. Este algoritmo es particularmente útil cuando quiera buscar un elemento que cumple ciertos criterios. Por ejemplo, si un contenedor contiene una lista de correo, podría usar **find_if()** para encontrar direcciones que tienen un código postal específico.

Tanto **find()** como **find_if()** pueden operar en un rango no ordenado. Esto significa que pueden usarse en cualquier tipo de contenedor y no hay necesidad de que se mantenga en orden. También funcionarán con un contenedor ordenado, pero existen mejores algoritmos de búsqueda para este tipo de contenedores. Consulte la sección *Opciones* en esta solución, para conocer un ejemplo.

Ejemplo

Con el siguiente ejemplo se ilustran **find()** y **find_if()**. Se usa un vector para contener cadenas. Luego se usa **find()** para encontrar la primera cadena que coincida con "dos". Luego se usa **find_if()** para encontrar una cadena que tiene tres caracteres o menos.

```
// Demuestra los algoritmos find() y find_if().

#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

bool es_cadena_corta(string cad);

int main()
{
  vector<string> v;
  vector<string>::iterator itr;

  v.push_back("uno");
  v.push_back("dos");
  v.push_back("tres");
  v.push_back("cuatro");
  v.push_back("cinco");
  v.push_back("seis");

  cout << "Contenido de v: ";
  for(unsigned i=0; i < v.size(); ++i)
    cout << v[i] << " ";
  cout << "\n\n";

  // Encuentra el elemento que contiene "dos".
  cout << "Buscando \"dos\"\n";
  itr = find(v.begin(), v.end(), "dos");
  if(itr != v.end()) {
    cout << "Se ha encontrado \"dos\", Reemplazando con \"DOS\"\n";
    *itr = "DOS";
  }
  cout << endl;

  // Encuentra todas las cadenas que tienen menos de cuatro caracteres de largo.
  cout << "Buscando todas las cadenas que tienen 3 o menos caracteres.\n";
  itr = v.begin();
  do {
    itr = find_if(itr, v.end(), es_cadena_corta);
    if(itr != v.end()) {
      cout << "Encontrado " << *itr << endl;
      ++itr;
    }
  } while(itr != v.end());
```

```
  return 0;
}

// Devuelve true si la cadena tiene tres caracteres o menos.
bool es_cadena_corta(string cad)
{
  if(cad.size() <= 3) return true;
  return false;
}
```

Aquí se muestra la salida:

```
Contenido de v: uno dos tres cuatro cinco seis

Buscando "dos"
Se ha encontrado "dos", Reemplazando con "DOS"

Buscando todas las cadenas que tienen tres o menos caracteres.
Encontrado uno
Encontrado DOS
Encontrado seis
```

En el programa, observe cómo se usa **find_if()** en un bucle para permitir que se encuentren todas las cadenas que tengan tres o menos caracteres. Cada búsqueda empieza donde se quedó la anterior. Esto es posible porque **find_if()** devuelve un iterador al elemento encontrado. Este iterador puede entonces aumentarse y usarse para empezar la búsqueda siguiente. Recuerde que **find()** y **find_if()** (y casi todos los demás algoritmos) operan en un rango específico de elementos, en lugar de hacerlo en todo lo que incluye el contenedor. Esto hace que estos algoritmos resulten mucho más versátiles de lo que de otra manera serían.

Ejemplo adicional: extraiga frases de un vector de caracteres

Aunque en el ejemplo anterior se presenta la mecánica de **find_if()**, no se muestra todo su potencial. Al crear con todo cuidado un predicado, puede usarse **find_if()** para realizar operaciones de búsqueda muy complejas. Por ejemplo, su predicado puede mantener información de estado que se usa para encontrar elementos con base en un contexto. En el siguiente programa se presenta un estudio de caso simple. Se usa **find_if()** para extraer frases de un vector de caracteres. Se utiliza un predicado llamado **es_inicio_frase()** para encontrar el inicio de cada frase. Esta función mantiene información de estado cuando se ha alcanzado el final de una frase.

```
// Extrae frases de un vector de caracteres con ayuda de find_if().

#include <iostream>
#include <vector>
#include <algorithm>
#include <cstring>
#include <cctype>

using namespace std;

bool es_inicio_frase(char car);

template<class InIter>
    void mostrar_rango(const char *msj, InIter start, InIter end);
```

```
int main()
{
  vector<char> v;
  vector<char>::iterator itr;
  const char *cad = "\u00a8Se trata de una prueba? \u00adS\u00a1, es una prueba!
  Como esta otra.";

  for(unsigned i=0; i < strlen(cad); i++)
    v.push_back(cad[i]);

  mostrar_rango("El contenido de v: ", v.begin(), v.end());
  cout << endl;

  // Encuentra el principio de todas las frases.
  cout << "Se usa find_if() para mostrar todas las frases de v:\n";

  // itr_inicio señalará al principio de la frase y
  // itr_final señalará al principio de la siguiente frase.
  vector<char>::iterator itr_inicio, itr_final;

  itr_inicio = v.begin();
  do {
    // Encuentra el inicio de una frase.
    itr_inicio = find_if(itr_inicio, v.end(), es_inicio_frase);

    // Encuentra el principio de la siguiente frase.
    itr_final = find_if(itr_inicio, v.end(), es_inicio_frase);

    // Muestra la secuencia intermedia.
    mostrar_rango("", itr_inicio, itr_final);
  } while(itr_final != v.end());

  return 0;
}

// Devuelve true si car es la primera letra de una frase.
bool es_inicio_frase(char car) {
  static bool findefrase = true;

  if((car) && findefrase ) {
    findefrase = false;
    return true;
  }

  if(car=='.' || car=='?' || car=='!') findefrase = true;
  return false;
}

// Muestra un rango de elementos.
template<class InIter>
    void mostrar_rango(const char *msj, InIter start, InIter end) {

  InIter itr;
```

```
  cout << msj;

  for(itr = start; itr != end; ++itr)
    cout << *itr;
  cout << endl;
}
```

Aquí se muestra la salida:

```
El contenido de v: ¿Se trata de una prueba? ¡Sí, es una prueba! Como esta otra.

Se usa find_if() para mostrar todas las frases de v:
¿Se trata de una prueba?
¡Sí, es una prueba!
Como esta otra.
```

Preste especial atención a la manera en que funciona el predicado **es_inicio_frase()**. Busca el primer carácter después del final de una frase anterior. Utiliza una **static bool** llamada **findefrase** para indicar que se ha encontrado el final de una frase. Se supone que se ha llegado a éste si se encuentra un carácter de terminación de frase (un punto o un signo de interrogación o admiración). En este caso, **findefrase** se establece como true. Cuando es true, entonces se supone que el siguiente carácter es el inicio de la siguiente frase. Cuando esto ocurre, **findefrase** se establece como false y **es_inicio_frase()** devuelve true. En todos los demás casos, devuelve false. Observe que es true al principio para que se encuentre la primera frase.

Opciones

Para buscar una secuencia de elementos, en lugar de un valor específico, utilice el algoritmo **search()**. Se describe en la siguiente solución.

Si está operando en una secuencia ordenada, entonces puede usar una búsqueda binaria para encontrar un valor. En casi todos los casos, este tipo de búsqueda es mucho más rápida que una secuencial. Por supuesto, requiere una secuencia ordenada. Aquí se muestran los prototipos de **binary_search()**:

template <class ForIter, class T>
  bool binary_search(ForIter *inicio*, ForIter *end*, const T &*val*)
template <class ForIter, class T, class Comp>
  bool binary_search(ForIter *inicio*, ForIter *end*, const T &*val*, Comp *fucomp*)

El algoritmo **binary_search** realiza una búsqueda binaria del valor especificado por *val* en un rango ordenado de *inicio* a *final*-1. Devuelve true si se encuentra *val*, y false, de otra manera. La primera versión compara los elementos de la secuencia especificada. La segunda versión le permite especificar su propia función de comparación. Cuando se actúa sobre iteradores de acceso aleatorio, **binary_search()** consume tiempo logarítmico. En el caso de otros tipos de iteradores, el número de comparaciones es logarítmico, aunque el tiempo que se toma moverse entre elementos no lo sea.

Podría sorprenderle el hecho de que **binary_search()** devuelve un resultado true/false en lugar de un iterador al elemento que encuentra. Una justificación para este método está basada en el argumento de que una secuencia ordenada puede contener dos o más valores que coinciden con el que se busca. Por tanto, hay poco valor en devolver el primero encontrado. La validez de este argumento ha estado sujeta a debate; no obstante, es la manera en que funciona **binary_search()**.

Para obtener en realidad un iterador a un elemento en una secuencia ordenada, utilizará uno de estos algoritmos: **lower_bound()**, **upper_bound()** o **equal_range()**. Los prototipos para las versiones que no son de predicado de estos algoritmos se muestran a continuación:

```
template <class ForIter, class T>
  pair<ForIter, ForIter> equal_range(ForIter inicio, ForIter final,
                                     const T &val)

template <class ForIter, class T>
  ForIter lower_bound(ForIter inicio, ForIter final, const T &val)

template <class ForIter, class T>
  ForIter upper_bound(ForIter inicio, ForIter final, const T &val)
```

El algoritmo **lower_bound()** devuelve un iterador al primer elemento que es igual o mayor que *val*, **upper_bound()** devuelve un iterador uno más allá del último elemento coincidente (en otras palabras, el primer elemento mayor que *val*) y **equal_range()** devuelve un par de iteradores que señalan a los límites inferior y superior. Todos estos algoritmos operan en tiempo logarítmico cuando actúan en iteradores de acceso aleatorio debido a que, además, usan una búsqueda binaria para encontrar sus valores respectivos. En el caso de otros tipos de iteradores, el número de comparaciones es logarítmico, aunque el tiempo que toma moverse entre elementos no lo sea. En general, si quiere obtener un iterador al primer elemento coincidente en una secuencia ordenada, use **equal_range()**. Si los iteradores de límite inferior y superior difieren, entonces sabrá que por lo menos se ha encontrado un elemento coincidente y el iterador del límite inferior señalará a la primera aparición del elemento.

Aunque encontrar un elemento específico suele ser lo que se necesita, en algunos casos, querrá encontrar la primera aparición de cualquier elemento de un conjunto. Una manera de hacer esto es usar el algoritmo **find_first_of()**, que se muestra aquí:

```
template <class ForIter, class ForIter2>
  ForIter find_first_of(ForIter1 inicio1, ForIter1 final1,
                        ForIter2 inicio2, ForIter2 final2)

template <class ForIter1, class ForIter2, class BinPred>
  ForIter find_first_of(ForIter1 inicio1, ForIter1 final1,
                        ForIter2 inicio2, ForIter2 final2,
                        BindPred funp)
```

Encuentra el primer elemento dentro de un rango *inicio1* a *final1*-1, que coincide con cualquier elemento dentro del rango *inicio2* a *final2*-1. Devuelve un iterador al elemento coincidente o *final1* si no se encuentra coincidencia. La segunda forma le permite especificar un predicado binario que determina cuando dos elementos son iguales.

Un algoritmo interesante que será útil en algunos casos es **adjacent_find()**. Busca la primera aparición de un par coincidente de elementos adyacentes. Aquí se muestran sus primeras dos versiones:

template <class ForIter> ForIter adjacent_find(ForIter *inicio*, ForIter *final*)

template <class ForIter, class BinPred> ForIter adjacent_find(ForIter *inicio*, ForIter *final*,
BinPred *funp*)

El algoritmo **adjacent_find()** busca elementos coincidentes adyacentes dentro del rango *inicio* a *final*-1. Devuelve un iterador al primer elemento del primer par coincidente. Devuelve *fin* si no se encuentran elementos coincidentes adyacentes. La segunda forma le permite especificar un predicado binario que determina cuando dos elementos son iguales.

Otra variación interesante en la búsqueda es el algoritmo **mismatch()**, que le permite encontrar la primera falta de coincidencia entre dos secuencias. Aquí se muestra su prototipo:

template <class InIter1, class InIter2>
pair<InIter1, InIter2> mismatch(InIter1 *inicio1*, InIter1 *final1*, InIter2 *inicio2*)

template <class InIter1, class InIter2, class BinPred>
pair<InIter1, InIter2> mismatch(InIter1 *inicio1*, InIter1 *final1*,
InIter2 *inicio2*, BinPred *funp*)

El algoritmo **mismatch()** encuentra la primera falta de coincidencia entre los elementos en el rango *inicio1* a *final*-1, y el que empieza con *inicio2*. Se devuelven los iteradores a los dos elementos no coincidentes. Si no se encuentra una falta de coincidencia, entonces se devuelven los iteradores *último1* y *primero2* + (*último1-primero1*). Por tanto, es la longitud de la primera secuencia la que determina el número de elementos probados. La segunda forma le permite especificar un predicado binario que determina cuando un elemento es igual a otro. (La clase de plantilla **pair** contiene dos campos, llamados **first** y **second**, que contienen el par de iteradores. Consulte el capítulo 3 para conocer más detalles.)

Use search() para encontrar una secuencia coincidente

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template <class ForIter1, class ForIter2> ForIter1 search(ForIter1 *inicio1*, ForIter1 *final1*, ForIter2 *inicio2*, ForIter2 *final2*)

En la solución anterior se mostró cómo buscar un elemento específico. En esta solución se muestra cómo buscar una secuencia de elementos. Este tipo de búsqueda, obviamente, es muy útil en varias situaciones. Por ejemplo, suponga una **deque** que contiene cadenas que indican el éxito o la falla en los intentos por iniciar sesión en una red. Tal vez quiera revisar el contenedor para encontrar apariciones en que se ha ingresado una contraseña incorrecta tres veces en fila, que podría indicar un intento de irrupción. Para ello, necesita buscar una secuencia de tres fallos. La búsqueda de una sola falla no es suficiente. El principal algoritmo que se usa para encontrar una secuencia es **search()**, y se demuestra en esta solución.

Paso a paso

Para buscar una secuencia de elementos se requieren estos pasos:

1. Defina la secuencia que desea encontrar.
2. Llame a **search(),** pasándolo en iteradores al inicio y el final del rango y de la secuencia en que se buscará.

Análisis

El algoritmo **search()** busca una secuencia de elementos. Tiene dos formas. Aquí se muestra la usada en esta solución:

```
template <class ForIter1, class ForIter2>
  ForIter1 search(ForIter1 inicio1, ForIter1 final1,
                  ForIter2 inicio2, ForIter2 final2)
```

La secuencia que se está buscando está definida por el rango *inicio1* a *final*-1. La subsecuencia que se está buscando está especificada por *inicio2* a *final2*-1. Si se encuentra, se devuelve un iterador a su principio. De otra manera, se devuelve *final1*.

No es obligatorio que las secuencias de búsqueda estén en el mismo tipo de contenedor. Por ejemplo, puede buscar una secuencia en una lista que coincide con una secuencia de un vector. Ésta es una de las ventajas de los algoritmos de STL. Debido a que funcionan mediante iteradores, pueden aplicarse a cualquier contenedor que dé soporte al tipo de iterador requerido, que es un iterador directo, en este caso.

Ejemplo

En el siguiente ejemplo se muestra **search()** en acción. Busca una **deque** que contiene una respuesta de inicio de sesión en red. Busca una serie de intentos de inicio de sesión en que se introdujo la contraseña incorrecta tres veces en fila, lo que podría indicar un posible ingreso indebido. Para este ejemplo, suponga que el registro de red puede contener varios tipos de respuesta, como inicio correcto, conexión fallida, etc. Sin embargo, cuando se introduce una contraseña incorrecta, las dos siguientes respuestas se colocan en el registro:

contraseña no válida
reingrese contraseña

Para buscar posibles intentos de ingreso indebido, el programa busca casos en que estas respuestas ocurren tres veces en fila. Si encuentra esta respuesta, informa que ha ocurrido un posible intento indebido de ingreso en la red.

NOTA *Otro ejemplo del algoritmo* ***search()*** *se encuentra en el capítulo 2, en la solución* Cree una búsqueda no sensible a mayúsculas y minúsculas y funciones de búsqueda y reemplazo para objetos **string**.

```
// Demuestra search().

#include <iostream>
#include <deque>
#include <algorithm>
#include <string>

using namespace std;

int main()
{
  deque<string> registro;
  deque<string> ingreso_indebido;
  deque<string>::iterator itr;

  // Crea una secuencia de tres respuestas de contraseña no válida.
  ingreso_indebido.push_back("contrase\u00a4a no v\u00a0lida");
  ingreso_indebido.push_back("reingrese contrase\u00a4a ");
  ingreso_indebido.push_back("contrase\u00a4a no v\u00a0lida");
  ingreso_indebido.push_back("reingrese contrase\u00a4a ");
  ingreso_indebido.push_back("contrase\u00a4a no v\u00a0lida");


  // Crea algunas entradas de registro.
  registro.push_back("inicio correcto");
  registro.push_back("contrase\u00a4a no v\u00a0lida");
  registro.push_back("reingrese contrase\u00a4a ");
  registro.push_back("inicio correcto");
  registro.push_back("conexi\u00a2n fallida");
  registro.push_back("inicio correcto");
  registro.push_back("inicio correcto");
  registro.push_back("contrase\u00a4a no v\u00a0lida");
  registro.push_back("reingrese contrase\u00a4a ");
  registro.push_back("contrase\u00a4a no v\u00a0lida");
  registro.push_back("reingrese contrase\u00a4a ");
  registro.push_back("contrase\u00a4a no v\u00a0lida");
  registro.push_back("conflicto en puerto");
  registro.push_back("inicio correcto");

  cout << "El registro:\n";
  for(itr = registro.begin(); itr != registro.end(); ++itr)
    cout << *itr << endl;
  cout << endl;

  // Ve si se hizo un intento indebido.
  itr = search(registro.begin(), registro.end(), ingreso_indebido.begin(),
  ingreso_indebido.end());

  if(itr != registro.end())
```

```
    cout << "Se ha encontrado un posible intento de ingreso indebido.\n";
  else
    cout << "No se encontraron fallas repetidas de contrase\u00a4a.\n";

  return 0;
}
```

Aquí se muestra la salida:

```
El registro:
inicio correcto
contraseña no válida
reingrese contraseña
inicio correcto
conexión fallida
inicio correcto
inicio correcto
contraseña no válida
reingrese contraseña
contraseña no válida
reingrese contraseña
contraseña no válida
conflicto en puerto
inicio correcto

Se ha encontrado un posible intento de ingreso indebido.
```

Es importante comprender que la llamada a **search()** sólo tendrá éxito si ocurren tres respuestas de contraseña no válida en fila. Para confirmarlo, trate de convertir en comentarios una de las llamadas a **registro.push_back("reingrese contrase\u00a4a ")**. Cuando se ejecute el programa, ya no encontrará una secuencia coincidente.

Opciones

Hay una segunda forma de **search()** que le permite especificar un predicado binario que determina cuando dos elementos son iguales. Aquí se muestra:

template <class ForIter1, class ForIter2, class BinPred>
ForIter1 search(ForIter1 *inicio1*, ForIter1 *final1*,
ForIter2 *inicio2*, ForIter2 *final2*, BinPred *funp*)

Funciona de la misma manera que la primera versión, con la excepción de que el predicado binario se pasa en *fun.*

Puede encontrar la última aparición de una secuencia al llamar a **find_end()**, que se muestra aquí:

template <class ForIter1, class ForIter2>
ForIter1 find_end(ForIter1 *inicio1*, ForIter1 *final1*,
ForIter2 *inicio2*, ForIter2 *final2*)

template <class ForIter1, class ForIter2, class BinPred>
ForIter1 find_end(ForIter1 *inicio1*, ForIter1 *final1*,
ForIter2 *inicio2*, ForIter2 *final2*,
BinPred *funp*)

Funciona de la misma manera que **search()**, con la excepción de que encuentra la última aparición, en lugar de la primera, en el rango especificado por *inicio2* y *final2*, dentro del rango especificado por *inicio1* a *final1*.

Para buscar una secuencia de una longitud especificada en que todos los valores son los mismos, considere el uso de **search_n()**. Tiene dos formas, que se muestran aquí:

```
template <class ForIter1, class Size, class T>
  ForIter1 search_n(ForIter1 inicio, ForIter1 final,
                    Size num, const T &val)

template <class ForIter1, class Size, class T, class BinPred>
  ForIter1 search_n(ForIter1 inicio, ForIter1 final,
                    Size num, const T &val, BinPred funp)
```

Dentro del rango *inicio* a *final*-1, **search_n()** busca una secuencia de *num* elementos que son iguales a *val*. Si se encuentra la secuencia, se devuelve un iterador a su principio. De otra manera, se devuelve *final*. La segunda forma le permite especificar un predicado binario que determina cuando un elemento es igual a otro.

Otros algoritmos que se relacionan con la búsqueda de una secuencia son **equal()**, que compara la igualdad entre dos secuencias, y **mismatch()**, que encuentra la primera falta de coincidencia entre dos secuencias.

Invierta, gire y modifique el orden de una secuencia

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template <class RandIter> void random_shuffle(RandIter *inicio*, RandIter *final*) template <class BiIter> void reverse(BiIter *inicio*, BiIter *final*) template <class ForIter> void rotate(ForIter *inicio*, ForIter *mid*, ForIter *final*)

En esta solución se demuestra el uso de tres algoritmos relacionados: **reverse()**, **rotate()** y **random_shuffle()**. Se relacionan entre sí porque cada uno cambia el rango al que se aplica. El algoritmo **reverse()** invierte la secuencia, **rotate()** la gira (es decir, toma un elemento de un extremo y lo coloca en el otro) y **random_shuffle()** dispone en orden aleatorio los elementos.

Paso a paso

Para invertir, girar o "barajar" una secuencia, se requieren estos pasos:

1. Invierta una secuencia al llamar a **reverse()**, especificando los extremos del rango que se habrá de invertir.
2. Gire una secuencia al llamar a **rotate()**, especificando los extremos del rango que se habrá de girar.
3. Disponga en orden aleatorio los elementos dentro de una secuencia al llamar a **random_shuffle()**, especificando los extremos del rango que se habrá de barajar.

Análisis

Puede invertir el contenido de una secuencia al llamar a **reverse()**. Tiene el prototipo:

template <class BiIter> void reverse(BiIter *inicio*, BiIter *final*)

El algoritmo **reverse ()** invierte el orden del rango *inicio* a *final*-1.

El algoritmo **reverse()** realiza una rotación a la izquierda. Una **rotación** es un desplazamiento en que el valor se transfiere de un extremo al otro. El prototipo para **rotate()** se muestra aquí:

template <class ForIter>
void rotate(ForIter *inicio*, ForIter *medio*, ForIter *final*)

El algoritmo **rotate()** gira a la izquierda los elementos en el rango de *inicio* a *final*-1, de modo que el elemento especificado por *medio* se vuelve el primer elemento nuevo.

Un algoritmo particularmente útil para los programadores que crean simulaciones es **random_shuffle()**. Reordena los elementos en una secuencia de manera aleatoria. Tiene las dos versiones mostradas aquí:

template <class RandIter>
void random_shuffle(RandIter *inicio*, RandIter *final*)

template <class RandIter, class Generator>
void random_shuffle(RandIter *inicio*, RandIter *final*, Generator *gen_al*)

El algoritmo **random_shuffle()** distribuye en orden aleatorio el rango de *inicio* a *final*-1. En la segunda forma, *gen_al* especifica un generador de números aleatorios. Esta función debe tener la siguiente forma general:

gen_al(*num*)

Debe devolver un número aleatorio entre cero y *num*. Observe que **random_shuffle()** requiere iteradores de acceso aleatorio. Esto significa que puede usarse en contenedores como **vector** y **deque**, pero no **list**, por ejemplo.

Ejemplo

En el siguiente ejemplo se demuestran **reverse()**, **rotate()** y **random_shuffle()**:

```
// Invierte, gira y ordena de manera aleatoria una secuencia.

#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

void mostrar(const char *msj, vector<int> vect);

int main()
{
  vector<int> v;

  for(int i=0; i<10; i++) v.push_back(i);

  mostrar("Orden original: ", v);
  cout << endl;

  // Invierte v.
  reverse(v.begin(), v.end());
  mostrar("Tras invertir: ", v);
  cout << endl;

  // Invierte de nuevo para restaurar el orden original.
  reverse(v.begin(), v.end());
  mostrar("Tras la segunda llamada a reverse(): ", v);
  cout << endl;

  // Gira a la izquierda una posición.
  rotate(v.begin(), v.begin()+1, v.end());

  mostrar("Orden tras girar a la izquierda una posiciu\u00a2n:  ", v);
  cout << endl;

  // Ahora gira a la izquierda dos posiciones.
  rotate(v.begin(), v.begin()+2, v.end());

  mostrar("Orden tras girar a la izquierda dos posiciones: ", v);
  cout << endl;

  // Dispone v en orden aleatorio.
  random_shuffle(v.begin(), v.end());
  mostrar("Tras aplicar el orden aleatorio: ", v);

  return 0;
}

// Despliega el contenido de vector<int>.
void mostrar(const char *msj, vector<int> vect) {
  cout << msj;
  for(unsigned i=0; i < vect.size(); ++i)
    cout << vect[i] << " ";
  cout << "\n";
}
```

Aquí se muestra la salida:

```
Orden original: 0 1 2 3 4 5 6 7 8 9

Tras invertir: 9 8 7 6 5 4 3 2 1 0

Tras la segunda llamada a reverse(): 0 1 2 3 4 5 6 7 8 9

Orden tras girar a la izquierda una posición:  1 2 3 4 5 6 7 8 9 0

Orden tras girar a la izquierda dos posiciones: 3 4 5 6 7 8 9 0 1 2

Tras aplicar el orden aleatorio: 1 4 2 5 3 8 0 6 7 9
```

Ejemplo adicional: use iteradores inversos para realizar una rotación a la derecha

Aunque la STL proporciona un algoritmo de giro a la izquierda, no provee uno para girar a la derecha. Al principio, esto podría parecer un error serio en el diseño de STL, o por lo menos una omisión que podría causar problemas. Pero no es el caso. Para realizar una rotación a la derecha, use el algoritmo **rotate()**, pero llámelo usando iteradores inversos. Como éstos actúan en reversa, el efecto neto de este tipo de llamada es que se realiza una rotación a la derecha en la secuencia. Esta técnica se demuestra con el siguiente programa.

```
// Gira una secuencia a la derecha empleando iteradores inversos
// con el algoritmo rotate().

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void mostrar(const char *msj, vector<int> vect);

int main()
{
  vector<int> v;

  for(int i=0; i<10; i++) v.push_back(i);

  mostrar("Orden original: ", v);
  cout << endl;

  // Gira a la derecha dos posiciones empleando iteradores inversos.
  rotate(v.rbegin(), v.rbegin()+2, v.rend());

  mostrar("Orden tras dos rotaciones a la derecha: ", v);

  return 0;
}

// Despliega el contenido de vector<int>.
```

```
void mostrar(const char *msj, vector<int> vect) {
  cout << msj;
  for(unsigned i=0; i < vect.size(); ++i)
    cout << vect[i] << " ";
  cout << "\n";
}
He aquí la salida del programa:
Orden original: 0 1 2 3 4 5 6 7 8 9

Orden tras dos rotaciones a la derecha: 8 9 0 1 2 3 4 5 6 7
```

Como puede ver, la secuencia original se ha girado dos posiciones.

Como se ilustra con esta aplicación de **rotate()**, parte de la capacidad y la elegancia de STL proviene de las sutilezas de su diseño. Al definir iteradores inversos, los creadores de STL permitieron que varios algoritmos operaran en orden inverso, reduciendo así la necesidad de definir explícitamente un complemento de ejecución hacia atrás para cada algoritmo. Aunque hubiera sido posible crear una biblioteca de plantillas que no incluyeran cosas como iteradores inversos, es este tipo de construcciones lo que da elegancia a su diseño.

Opciones

Hay una variación de **reverse()** llamada **reverse_copy()** que le podría resultar útil en algunos casos. En lugar de invertir el contenido de la secuencia especificada, copia la secuencia invertida en otro rango. He aquí su prototipo:

template <class BiIter, class OutIter>
  void reverse_copy(BiIter *inicio*, BiIter *final*, OutIter *inicio_resultado*)

Copia en orden inverso el rango *inicio* a *final*-1 en la secuencia cuyo elemento inicial es señalado por *inicio_resultado*. El rango al que señala *inicio_resultado* debe ser por lo menos del mismo tamaño que el rango invertido.

De manera similar, hay una variación de **rotate()** llamada **rotate_copy()** que copia la secuencia girada en otro rango. Aquí se muestra:

template <class ForIter, class OutIter>
  void rotate_copy(ForIter *inicio*, ForIter *medio*, ForIter *final*, OutIter *inicio_resultado*)

Copia el rango *inicio* a *final*-1 en el rango cuyo primer elemento es señalado por *inicio_resultado*. El rango al que señala éste debe ser por lo menos del mismo tamaño que el rango girado. En el proceso, gira a la izquierda los elementos, de modo que el elemento especificado por *medio* se vuelve el primer elemento nuevo. Devuelve un iterador a uno después del final del rango resultante.

Puede crear permutaciones de un rango al llamar a **next_permutation()** o **prev_permutation()**. Se describen en *Permute una secuencia*.

Recorra en ciclo un contenedor con for_each()

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template<class InIter, class Func> Func for_each(InIter *inicio*, InIter *final*, Func *fn*)

Como casi todos los programadores saben, recorrer en ciclo el contenido de un contenedor es una actividad muy común. Por ejemplo, para desplegar el contenido de un contenedor, necesitará recorrerlo en ciclo del principio al final, desplegando cada elemento de uno en uno. Esta actividad puede realizarse de diferentes maneras. Por ejemplo, puede recorrer en ciclo cualquier tipo de contenedor mediante el uso de un iterador. Contenedores como **vector** y **deque** le permiten recorrer en ciclo sus contenedores mediante el operador de subíndice de matriz. El algoritmo **for_each** ofrece otro método de hacerlo. Recorre en ciclo un rango de elementos, aplicando una operación específica a cada uno. En esta solución se demuestra su uso.

Paso a paso

Para recorrer en ciclo un rango de elementos, mediante el uso de **for_each(),** se requieren estos pasos:

1. Cree una función (o un objeto de función) que será llamado por cada elemento en el rango.
2. Llame a **for_each()**, pasando iteradores al principio y al final del rango que habrá de procesarse y la función que habrá de aplicarse.

Análisis

Aquí se muestra el prototipo para el algoritmo **for_each()**:

template<class InIter, class Func>
 Func for_each(InIter *inicio*, InIter *final*, Func *fn*)

El algoritmo **for_each()** aplica la función *fn* al rango de elementos especificado por *inicio* a *final*. Por tanto, cada elemento del rango llama una vez a *fn*. **for_each()** devuelve *fn*. Puede pasar un apuntador a función o un objeto de función a *fn*. En ambos casos, *fn* debe tomar un argumento cuyo tipo sea compatible con el de los elementos en el rango especificado. Puede devolver un valor. Sin embargo, si *fn* devuelve un valor, éste es ignorado por **for_each()**. Por tanto, con frecuencia el tipo devuelto por *fn* es **void**. Sin embargo, un valor devuelto podría ser útil en situaciones en que no se llama al algoritmo **for_each()**. Por ejemplo, *fn* podría mantener una cuenta del número de elementos que se procesa y devolver esta cuenta después de la que devuelve el algoritmo **for_each()**.

Ejemplo

En el siguiente ejemplo se muestra **for_each()** en acción. Se utiliza para dos propósitos. En primer lugar, una llamada a **for_each()** despliega el contenido de un contenedor, de elemento en elemento. Utiliza la función **mostrar()** para desplegar cada elemento. En segundo lugar, calcula la suma de los elementos del contenedor. En este caso, **for_each()** pasa un apuntador a la función **sumatoria()**. Observe que esta función devuelve la sumatoria. Este valor no es usado por **for_each()**. En cambio, se obtiene después de lograr la suma de los elementos.

```
// Demuestra el algoritmo for_each().
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Despliega un valor int.
void mostrar(int i) {
  cout << i << " ";
}

// Mantiene una suma actualizada de los valores pasados a i.
int sumatoria(int i) {
  static int suma = 0;

  suma += i;
  return suma;
}

int main()
{
  vector<int> v;
  int i;

  for(i=1; i < 11; i++) v.push_back(i);

  cout << "Contenido de v: ";
  for_each(v.begin(), v.end(), mostrar);
  cout << "\n";

  for_each(v.begin(), v.end(), sumatoria);
  cout << "Sumatoria de v: " << sumatoria(0);

  return 0;
}
```

Aquí se muestra la salida:

```
Contenido de v: 1 2 3 4 5 6 7 8 9 10
Sumatoria de v: 55
```

Como se explicó en el análisis, la función pasada a **for_each()** debe tener un parámetro, y el tipo de éste debe ser el mismo que el de elementos en el contenedor en que se usa **for_each()**. En este ejemplo, como **v** es un vector de **int**, **mostrar()** y **sumatoria()** tienen un parámetro **int**. Cada vez que se llama a una de estas funciones, se pasa un elemento del rango especificado. Debe

destacarse que la función **sumatoria()** es muy limitada. Una mejor manera de implementarlo es como un objeto de función, como se muestra en *Cree un objeto de función personalizado.*

Opciones

El estándar internacional de C++ ordena en categorías **for_each()** como un *algoritmo que no modifica*. Sin embargo, esta etiqueta puede llevar un poco a equívocos. Por ejemplo, no hay nada que evite que la función pasada a **for_each()** use un parámetro de referencia y modifique el elemento mediante la referencia. En otras palabras, una función aplicada a cada elemento en un contenedor debe declararse así:

void *fn*(*tipo* &*arg*)

En este caso, *arg* es un parámetro de referencia. Por tanto, el valor al que señala *arg* podría cambiarse mediante una asignación, como se muestra aquí:

```
arg = nuevovalor;
```

Por ejemplo, la siguiente función invertirá un carácter que se pasa, cambiando una mayúscula en minúscula y viceversa. Observe que se pasa **car** como referencia.

```
// Invierte las mayúsculas y minúsculas de un carácter pasado en car.
void inv_mayus(char &car) {
  if(islower(car)) car = toupper(car);
  else car = tolower(car);
}
```

Por tanto, suponiendo un vector llamado **v** que contiene caracteres, la siguiente llamada a **for_each()** modificará **v** de modo tal que cada carácter del contenedor vea invertidas sus mayúsculas y minúsculas.

```
for_each(v.begin(), v.end(), inv_mayus);
```

Aunque el código anterior funciona, el autor no se siente muy cómodo con él por dos razones. En primer lugar, como se explicó, el estándar internacional para C++ ordena **for_each()** como un algoritmo que no modifica. Aunque técnicamente no se rompe esta regla (debido a que el algoritmo, *en sí*, no modifica la secuencia), el cambio del contenido del contenedor como un efecto colateral de la función pasada a **for_each()** parece inconsistente y erróneo. En segundo lugar, STL ofrece una mejor manera de modificar una secuencia que utiliza el algoritmo **transform()**, que se describe en *Use **transform()** para cambiar una secuencia*.

El programa de ejemplo pasó un apuntador a función a **for_each()**, pero también puede pasar un objeto de función. Recuerde que un objeto de función es una instancia de una clase que implementa **operator()**. Los objetos de función se describen de manera detallada en *Use un objeto de función integrado* y *Cree un objeto de función personalizado*. Para conocer un ejemplo que use un objeto de función con **for_each()**, consulte *Cree un objeto de función personalizado*.

Use transform() para cambiar una secuencia

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template <class InIter, class OutIter, class Func> OutIter transform(InIter *inicio*, InIter *final*, OutIter *resultado*, Func *funcunaria*) template <class InIter1, class InIter2, class OutIter, class Func> OutIter transform(InIter1 *inicio1*, InIter1 *final1*, InIter2 *inicio2*, OutIter *resultado*, Func *funcbinaria*)

En ocasiones, querrá aplicar una transformación a todos los elementos dentro de una secuencia y almacenar el resultado. La mejor manera de lograr esto consiste en usar el algoritmo **transform()**. Tiene dos formas. La primera le permite aplicar una transformación a un rango de elementos de una sola secuencia. La segunda, aplicar una transformación a elementos de dos secuencias. En ambos casos, se almacena la secuencia resultante. Un aspecto clave de **transform()** es que la secuencia resultante puede ser la misma que la secuencia de entrada o puede ser diferente. Por tanto, **transform()** puede usarse para cambiar los elementos de una secuencia o para crear una secuencia separada que contiene el resultado. En esta solución se muestra el proceso.

Paso a paso

Para aplicar **transform()** a los elementos de un solo rango se requieren los siguientes pasos:

1. Cree una función (u objeto de función) que realice la transformación deseada. Debe tener un solo parámetro que reciba un elemento de un rango de entrada.
2. Llame a **transform()**, especificando el rango de entrada, el de salida y la función de transformación.

Para aplicar **transform()** a pares de elementos de dos rangos se requieren los siguientes pasos:

1. Cree una función (u objeto de función) que realice la transformación deseada. Debe tener dos parámetros, y cada uno recibe un elemento de un rango de entrada.
2. Llame a **transform()**, especificando ambos rangos de entrada, y la función de transformación.

Análisis

El algoritmo **transform()** tiene estas dos formas:

template <class InIter, class OutIter, class Func>
OutIter transform(InIter *inicio*, InIter *final*, OutIter *resultado*, Func *funcunaria*)

template <class InIter1, class InIter2, class OutIter, class Func>
OutIter transform(InIter1 *inicio1*, InIter1 *final1*, InIter2 *inicio2*,
OutIter *resultado*, Func *funcbinaria*)

El algoritmo **transform()** aplica una función a un rango de elementos y almacena la salida en *resultado*. El rango al que señala *resultado* debe tener por lo menos el tamaño del rango que se está transformando. En la primera forma, el rango está especificado por *inicio* a *final*. La función que se aplicará está especificada por *funcunaria*. Recibe el valor de un elemento en su parámetro y debe devolver su transformación. En la segunda forma de **transform()**, la transformación se aplica usando una función que recibe el valor de un elemento de la secuencia que habrá de transformarse (*inicio1* a *final1*) en su primer parámetro y un elemento de la segunda secuencia (empezando en *inicio2*) como segundo parámetro. Ambas versiones de **transform()** devuelven un iterador al final de la secuencia resultante.

Un aspecto clave de **transform()** es que puede usarse para cambiar el contenido de una secuencia en el lugar. Por tanto, para la primera forma de **transform()**, *resultado* e *inicio* pueden especificar el mismo elemento. Para la segunda forma, el resultado puede ser el mismo que *inicio1* o *inicio2*.

Hay un tema importante relacionado con **transform()**: el estándar internacional de C++ establece que la función de transformación (*funcunaria* o *funcbinaria*) no debe producir efectos colaterales.

Ejemplo

En el siguiente ejemplo se muestran ambas formas de **transform()** en acción. La primera se usa para calcular los recíprocos de una secuencia de valores **double** que se conservan en un vector. Esta transformación se aplica dos veces. En primer lugar, almacena los resultados en la secuencia original. La segunda vez, los almacena en otra secuencia. En ambos casos, la función **reciprocal()** se pasa a **transform()**.

La segunda forma de **transform()** calcula los puntos medios entre dos valores enteros contenidos en dos secuencias. Almacena el resultado en una tercera secuencia. La función **puntomedio()** realiza el cálculo del punto medio, y es la función que se pasa a **transform()**.

```
// Demuestra el algoritmo transform().
//
// Ambas versiones de transform() se usan dentro del
// programa. La primera altera la secuencia de doubles
// para que contenga valores recíprocos. La segunda
// crea una secuencia que contiene los puntos medios
// entre los valores en otras dos secuencias.

#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

double reciproco(double val);
int puntomedio(int a, int b);

template<class T> void mostrar(const char *msj, vector<T> vect);

int main()
{
  int i;

  // Primero, se demuestra la forma de una secuencia de transform().
  vector<double> v;

  // Coloque valores en v.
  for(i=1; i < 10; ++i) v.push_back((double)i);

  cout << "Demuestra la forma de una sola secuencia de transform().\n";
  mostrar("Contenido inicial de v:\n", v);
  cout << endl;

  // Transforma v al aplicar la función recíproco().
  // Coloca de nuevo el resultado en v.
  cout << "Calcula rec\u00a1procos para v y almacena los resultados en v.\n";
  transform(v.begin(), v.end(), v.begin(), reciproco);

  mostrar("Contenido transformado de v:\n", v);

  // Transforma v por segunda vez, colocando el resultado en una nueva secuencia.
  cout << "Transforma v de nuevo. Esta vez almacena los resultados en v2.\n";
  vector<double> v2(10);
  transform(v.begin(), v.end(), v2.begin(), reciproco);

  mostrar("Esto es v2:\n", v2);
  cout << endl;

  // Ahora, demuestra la forma de dos secuencias de transform()
  cout << "Demuestra la forma de dos secuencias de transform().\n";
  vector<int> v3, v4, v5(10);
  for(i = 0; i < 10; ++i) v3.push_back(i);
  for(i = 10; i < 20; ++i) if(i%2) v4.push_back(i); else v4.push_back(-i);

  mostrar("Contenido de v3:\n", v3);
  mostrar("Contenido de v4:\n", v4);
  cout << endl;

  cout << "Calcula puntos medios entre v3 y v4 y almacena los resultados en v5.\n";
  transform(v3.begin(), v3.end(), v4.begin(), v5.begin(), puntomedio);

  mostrar("Contenido de v5:\n", v5);

  return 0;
}
```

```
// Despliega el contenido de un vector<int>.
template<class T> void mostrar(const char *msj, vector<T> vect) {
  cout << msj;
  for(unsigned i=0; i < vect.size(); ++i)
    cout << vect[i] << " ";
  cout << "\n";
}

// Devuelve el punto medio entero entre dos valores.
int puntomedio(int a, int b) {
  return((a-b) / 2) + b;
}

// Devuelve el recíproco de un double.
double reciproco(double val) {
  if(val == 0.0) return 0.0;
  return 1.0 / val; // devuelve reciproco
}
```

Aquí se muestra la salida:

```
Demuestra la forma de una sola secuencia de transform().
Contenido inicial de v:
1 2 3 4 5 6 7 8 9

Calcula recíprocos para v y almacena los resultados en v.
Contenido transformado de v:
1 0.5 0.333333 0.25 0.2 0.166667 0.142857 0.125 0.111111
Transforma v de nuevo. Esta vez almacena los resultados en v2.
Esto es v2:
1 2 3 4 5 6 7 8 9 0

Demuestra la forma de dos secuencias de transform().
Contenido de v3:
0 1 2 3 4 5 6 7 8 9
Contenido de v4:
-10 11 -12 13 -14 15 -16 17 -18 19

Calcula puntos medios entre v3 y v4 y almacena los resultados en v5.
Contenido de v5:
-5 6 -5 8 -5 10 -5 12 -5 14
```

Un punto clave ilustrado por el programa es que la función o el objeto de función usado por **transform()** debe especificar un parámetro o varios parámetros cuyos tipos son compatibles con los de los elementos en las secuencias. Además, debe devolver un tipo compatible.

Opciones

No es necesario que el rango especificado en la versión de dos secuencias de **transform()** esté en dos contenedores separados. Se trata de un error de comprensión. En cambio, debe especificar ambos rangos desde el mismo contenedor. Por ejemplo, suponiendo el programa anterior, lo siguiente calcula los puntos medios entre los primeros y últimos cinco elementos de **v3** y los almacena en los primeros cinco elementos de **v5**:

```
transform(v3.begin(), v3.begin()+5, v3.begin()+5, v5.begin(), puntomedio();
```

Como ya se mencionó, es posible almacenar el resultado de nuevo en una de las secuencias originales, lo que permite que una secuencia se modifique en el lugar. Cuando se usa la forma de dos secuencias de **transform()**, la secuencia de destino puede ser cualquiera de las de entrada. Por ejemplo, esta instrucción calcula los puntos medios de las secuencias contenidas en **v3** y **v4** y almacena el resultado en **v4**:

```
transform(v3.begin(), v3.end(), v4.begin(), v4.begin(), puntomedio();
```

Esto funciona debido a que los valores de cada par de elementos se obtienen primero de cada secuencia y luego se pasan a **puntomedio()**. El resultado se almacena después en **v4**. Por tanto, los valores originales en **v4** se obtienen antes de que se sobrescriban.

En el ejemplo anterior se pasaron apuntadores de función a **transform()**, pero también puede usar objetos de función. Éstos se describen de manera detallada en *Use un objeto de función integrado* y *Cree un objeto de función personalizado*. Para conocer un ejemplo que usa un objeto de función con **transform()**, consulte *Cree un objeto de función personalizado*.

Si quiere realizar una operación sin modificación en una secuencia, considere el uso de **for_each()**. Consulte *Recorra en ciclo un contenedor con **for_each().***

En algunos casos, tal vez quiera generar una secuencia de elementos que no sean transformaciones de otra secuencia. Para ello, puede usar los algoritmos **generate()** o **generate_n()**. Aquí se muestran:

template <class ForIter, class Generator>
void generate(ForIter *inicio*, ForIter *final*, Generator *fungen*)

template <class ForIter, class Size, class Generator>
void generate_n(OutIter *inicio*, Size *num*, Generator *fungen*)

Los algoritmos **generate()** y **generate_n()** asignan valores devueltos por una función generadora a elementos dentro de un rango específico. En **generate()**, el rango que se está asignando está especificado por *inicio* a *final*. En el caso de **generate_n()**, el rango empieza en *inicio* y se ejecuta por *num* elementos. La función generadora se pasa en *fungen*. No tiene parámetros y debe devolver objetos que son compatibles con el tipo de la secuencia deseada. He aquí un ejemplo muy simple que demuestra **generate()**. Utiliza una función llamada **pot_de_dos()** para generar una secuencia que contiene una potencia de 2.

```
// Genera una secuencia.
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

double pot_de_dos();
```

```
int main()
{
  vector<double> v(5);

  // Genera una secuencia.
  generate(v.begin(), v.end(), pot_de_dos);

  cout << "Potencias de 2: ";
  for(unsigned i=0; i < v.size(); ++i)
    cout << v[i] << " ";

  return 0;
}

// Una función generadora simple que genera las potencias de 2.
double pot_de_dos() {
  static double val = 1.0;
  double t;

  t = val;
  val += val;

  return t;
}
```

Se despliega la siguiente salida:

```
Potencias de 2: 1 2 4 8 16
```

Realice operaciones con conjuntos

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template <class InIter1, class InIter2, class OutIter> OutIter set_union(InIter1 *inicio1*, InIter1 *final1*, InIter2 *inicio2*, InIter2 *final2*, OutIter *resultado*) template <class InIter1, class InIter2, class OutIter> OutIter set_difference(InIter1 *inicio1*, InIter1 *final1*, InIter2 *inicio2*, InIter2 *final2*, OutIter *resultado*) template <class InIter1, class InIter2, class OutIter> OutIter set_symmetric_difference(InIter1 *inicio1*, InIter1 *final1*, InIter2 *inicio2*, InIter2 *final2*, OutIter *resultado*) template <class InIter1, class InIter2, class OutIter> OutIter set_intersection(InIter1 *inicio1*, InIter1 *final1*, InIter2 *inicio2*, InIter2 *final2*, OutIter *resultado*) template <class InIter1, class InIter2> bool includes(InIter1 *inicio1*, InIter1 *final1*, InIter2 *inicio2*, InIter2 *final2*)

La STL proporciona cinco algoritmos que realizan operaciones con conjuntos. Debe comprenderse que estos algoritmos operan sobre cualquier tipo de contenedor; no se usan sólo con las clases **set** o **multiset**. El requisito es que el contenido del contenedor debe estar ordenado. Estos algoritmos de conjuntos son **set_union**, **set_difference()**, **set_symmetric_difference()**, **set_intersection()** e **includes()**. En esta solución se demuestra su uso.

Paso a paso

Para usar los algoritmos de conjuntos se requieren estos pasos:

1. Las dos secuencias que participarán en los algoritmos de conjunto deben ordenarse. Ambos deben contener elementos del mismo tipo o de tipos compatibles.
2. Obtenga la unión de los dos conjuntos al llamar a **set_union()**.

3. Obtenga la diferencia entre dos conjuntos al llamar a **set_difference()**.
4. Obtenga la diferencia simétrica entre dos conjuntos al llamar a **set_symmetric_difference()**.
5. Obtenga la intersección de dos conjuntos al llamar a **set_intersection()**.
6. Determine si un conjunto incluye otro conjunto completo al llamar a **includes()**. Este algoritmo puede usarse para determinar una relación de subconjunto.

Análisis

Para obtener la unión de dos conjuntos ordenados, se usa **set_union()**. Tiene dos formas. Aquí se muestra la usada en esta solución:

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_union(InIter1 inicio1, InIter1 final1,
          InIter2 inicio2, InIter2 final2, OutIter resultado)
```

Produce una secuencia que contiene la unión de los dos conjuntos definidos por los rangos *inicio1* a *final*-1 e *inicio2* a *final2*-1. Por tanto, el conjunto resultante contiene los elementos que se encuentran en ambos conjuntos. El resultado se ordena y se coloca en *resultado*. Los rangos de entrada no deben superponerse al rango resultante. Se devuelve un iterador al final del rango resultante.

Para obtener la diferencia entre dos conjuntos ordenados, se utiliza **set_difference()**. Tiene dos formas. Aquí se muestra la usada en esta solución:

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_difference(InIter1 inicio1, InIter1 final1,
          InIter2 inicio2, InIter2 final2, OutIter resultado)
```

El algoritmo **set_difference()** produce una secuencia que contiene la diferencia entre los dos conjuntos definidos por los rangos *inicio1* a *final*-1 e *inicio2* a *final2*-1. Es decir, el conjunto definido por *inicio2*, *final2* se elimina del conjunto definido por *inicio1*, *final1*. El resultado se ordena y se coloca en *resultado*. Los rangos de entrada no deben superponerse al rango resultante. Se devuelve un iterador al final del rango resultante.

La diferencia simétrica de dos conjuntos ordenados puede encontrarse empleando el algoritmo **set_symmetric_difference**. Tiene dos formas. Aquí se muestra la usada en esta solución:

```
template <class InIter1, class InIter2, class OutIter>
  OutIter set_symmetric_difference(InIter1 inicio1, InIter1 final1,
          InIter2 inicio2, InIter2 final2, OutIter resultado)
```

El algoritmo **set_symmetric_difference()** produce una secuencia que contiene la diferencia simétrica entre los dos conjuntos ordenados definidos por los rangos *inicio1* a *final*-1 e *inicio2* a *final2*-1. La diferencia simétrica de dos conjuntos sólo contiene los elementos que no son comunes para ambos conjuntos. El resultado se ordena y se coloca en *resultado*. Los rangos de entrada no deben superponerse al rango resultante. Se devuelve un iterador al final del rango resultante.

La intersección de dos conjuntos ordenados puede obtenerse al llamar a **set_intersection()**. Tiene dos formas. Aquí se muestra la usada en esta solución:

template <class InIter1, class InIter2, class OutIter>
OutIter set_intersection(InIter1 *inicio1*, InIter1 *final1*,
InIter2 *inicio2*, InIter2 *final2*, OutIter *resultado*)

El algoritmo **set_intersection()** produce una secuencia que contiene la intersección de los dos conjuntos definidos por los rangos *inicio* a *final*-1 e *inicio2* a *final2*-1. Son los elementos comunes para ambos conjuntos. El resultado se ordena y se coloca en *resultado*. Los rangos de entrada no deben superponerse al rango resultante. Se devuelve un iterador al final del rango resultante.

En el caso de todos los algoritmos anteriores, el rango señalado por *resultado* debe tener el largo suficiente para contener los elementos que se almacenarán en él. Los algoritmos de conjunto sobrescriben los elementos existentes. No insertan nuevos elementos.

Para ver si todo el contenido de un conjunto ordenado está incluido en otro, use **includes()**. Tiene dos formas. Aquí se muestra la usada en esta solución:

template <class InIter1, class InIter2>
bool includes(InIter1 *inicio1*, InIter1 *final1*,
InIter2 *inicio2*, InIter2 *final2*)

El algoritmo **includes()** determina si el rango *inicio* a *final*-1 incluye todos los elementos en el rango *inicio2* a *final2*-1. Devuelve true si se encuentran los elementos, y false, de otra manera. El algoritmo **includes()** puede usarse para determinar si un conjunto es un subconjunto de otro.

Recuerde que los algoritmos de conjunto pueden usarse con cualquier secuencia ordenada, no sólo instancias de **set** o **multiset**. Sin embargo, en todos los casos, la secuencia debe estar ordenada.

Ejemplo

Con el siguiente programa se demuestran todos los algoritmos de conjunto:

```
// Demuestra los algoritmos de conjuntos.
//
// Este programa usa list, pero puede usarse
// cualquier otro contenedor de secuencias.

#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

template<class InIter>
    void mostrar_rango(const char *msj, InIter start, InIter end);

int main()
{
  list<char> lista1, lista2, resultado(15), lista3;
  list<char>::iterator res_end;

  for(int i=0; i < 5; i++) lista1.push_back('A'+i);
  for(int i=3; i < 10; i++) lista2.push_back('A'+i);

  mostrar_rango("Contenido de lista1: ", lista1.begin(), lista1.end());
```

```
  cout << endl;

  mostrar_rango("Contenido de lista2: ", lista2.begin(), lista2.end());
  cout << endl;

  // Create the union of lista1 and lista2.
  res_end = set_union(lista1.begin(), lista1.end(),
                      lista2.begin(), lista2.end(),
                      resultado.begin());

  mostrar_rango("Uni\u00a2n de lista1 y lista2: ", resultado.begin(), res_end);
  cout << endl;

  // Crea un conjunto que contiene lista1 - lista2.
  res_end = set_difference(lista1.begin(), lista1.end(),
                           lista2.begin(), lista2.end(),
                           resultado.begin());

  mostrar_rango("lista1 - lista2: ", resultado.begin(), res_end);
  cout << endl;

  // Crea la diferencia simétrica entre lista1 y lista2.
  res_end = set_symmetric_difference(lista1.begin(), lista1.end(),
                                     lista2.begin(), lista2.end(),
                                     resultado.begin());

  mostrar_rango("Diferencia simétrica entre lista1 y lista2: ",
               resultado.begin(), res_end);
  cout << endl;

  // Crea la intersección entre lista1 y lista2.
  res_end = set_intersection(lista1.begin(), lista1.end(),
                             lista2.begin(), lista2.end(),
                             resultado.begin());

  mostrar_rango("Intersecci\u00a2n entre lista1 y lista2: ", resultado.begin(),
  res_end);
  cout << endl;

  // Usa includes() para revisar el subconjunto.
  lista3.push_back('A');
  lista3.push_back('C');
  lista3.push_back('D');

  if(includes(lista1.begin(), lista1.end(),
              lista3.begin(), lista3.end()))
    cout << "lista3 es un subconjunto de lista1\n";
  else
    cout << "lista3 no es un subconjunto de lista1\n";

  return 0;
}
```

```
// Muestra un rango de elementos.
template<class InIter>
    void mostrar_rango(const char *msj, InIter start, InIter end) {

  InIter itr;

  cout << msj;

  for(itr = start; itr != end; ++itr)
    cout << *itr << " ";
  cout << endl;
}
```

Este programa genera la siguiente salida:

```
Contenido de lista1: A B C D E

Contenido de lista2: D E F G H I J

Unión de lista1 y lista2: A B C D E F G H I J

lista1 - lista2: A B C

Diferencia simétrica entre lista1 y lista2: A B C F G H I J

Intersección entre lista1 y lista2: D E

lista3 es un subconjunto de lista1
```

Opciones

Todos los algoritmos de conjunto proporcionan una segunda forma que le permite especificar una función de comparación, lo que determina cuando un elemento es menor que otro. Puede usar esta función para especificar el orden de la secuencia de entrada y del resultado. Estas formas se muestran aquí:

template <class InIter1, class InIter2, class OutIter, class Comp>
 OutIter set_union(InIter1 *inicio1*, InIter1 *final1*,
  InIter2 *inicio2*, InIter2 *final2*, OutIter *resultado*, Comp *fucomp*)

template <class InIter1, class InIter2, class OutIter, class Comp>
 OutIter set_difference(InIter1 *inicio1*, InIter1 *final1*,
  InIter2 *inicio2*, InIter2 *final2*,
  OutIter *resultado*, Comp *fucomp*)

template <class InIter1, class InIter2, class OutIter, class Comp>
 OutIter set_symmetric_difference(InIter1 *inicio1*, InIter1 *final1*,
  InIter2 *inicio2*, InIter2 *final2*, OutIter *resultado*, Comp *fucomp*)

template <class InIter1, class InIter2, class OutIter, class Comp>
 OutIter set_intersection(InIter1 *inicio1*, InIter1 *final1*,
  InIter2 *inicio2*, InIter2 *final2*,
  OutIter *resultado*, Comp *fucomp*)

```
template <class InIter1, class InIter2, class Comp>
  bool includes(InIter1 inicio1, InIter1 final1,
                InIter2 inicio2, InIter2 final2, Comp fucomp)
```

Para todos los casos, los rangos especificados por *inicio1*, *final1* e *inicio2*, *final2* deben ordenarse de acuerdo con la función de comparación pasada en *fucomp*, que determina cuando un elemento es menor que otro. El resultado se ordenará de acuerdo con *fucomp*. De otra manera, estas funciones trabajan como sus versiones ya descritas.

Permute una secuencia

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template <class BiIter> bool next_permutation(BiIter *inicio*, BiIter *final*) template <class BiIter> bool prev_permutation(BiIter *inicio*, BiIter *final*)

Dos de los algoritmos más intrigantes son **next_permutation()** y **prev_permutation()**. Se usan para realizar permutaciones de una secuencia. Suelen usarse en simulaciones y en pruebas. Estos algoritmos requieren iteradores bidireccionales y sólo pueden usarse en secuencias que pueden ordenarse. En esta solución se demuestra su uso.

Paso a paso

Para permutar una secuencia se requieren estos pasos:

1. La secuencia que habrá de permutarse debe dar soporte a iteradores bidireccionales y permitir el ordenamiento.
2. Para obtener la siguiente permutación, llame a **next_permutation()**, especificando iteradores al principio y al final del rango que habrá de permutarse.
3. Para obtener la permutación anterior, llame a **prev_permutation()**, especificando iteradores al principio y al final del rango que habrá de permutarse.

Análisis

Puede generar una permutación de cualquier secuencia ordenada al usar los algoritmos **next_permutation()** y **prev_permutation()**. Cada una tiene dos formas. Aquí se muestran las usadas en esta solución:

template <class BiIter>
 bool next_permutation(BiIter *inicio*, BiIter *final*)
template <class BiIter>
 bool prev_permutation(BiIter *inicio*, BiIter *final*)

El algoritmo **next_permutation()** construye la siguiente permutación del rango *inicio* a *final*-1. El algoritmo **prev_permutation()** construye la permutación anterior del rango *inicio* a *final*-1. Las permutaciones se generan suponiendo que una secuencia ordenada representa la primera permutación. Si se han agotado todas las permutaciones, ambos algoritmos devuelven false. En este caso, **next_permutation()** organiza los rangos en orden ascendente y **prev_permutation** en descendente. De otra manera, ambas funciones devuelven true. Por tanto, un bucle que obtiene todas las permutaciones posibles se ejecutará hasta que se devuelva false.

Ejemplo

En el siguiente ejemplo se usa **next_permutation()** para generar todas las permutacione posibles de la secuencia ABC. Luego se usa **prev_permutation()** para generar las permutaciones a la inversa.

```
// Demuestra next_permutation() y prev_permutation().

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
  vector<char> v;
  unsigned i;

  // Esto crea la secuencia ordenada ABC.
  for(i=0; i<3; i++) v.push_back('A'+i);

  // Demuestra next_permutation().
  cout << "Todas las permutaciones de ABC con el uso de next_permutation():\n";
  do {
    for(i=0; i < v.size(); i++)
      cout << v[i];
    cout << "\n";
  } while(next_permutation(v.begin(), v.end()));

  // En este punto, v se ha vuelto a recorrer para contener ABC.

  cout << endl;

  // Demuestra prev_permutation().

  // En primer lugar, se respalda la primera permutación.
  prev_permutation(v.begin(), v.end());
```

```
  cout << "Todas las permutaciones de ABC con el uso de prev_permutation():\n";
  do {
    for(i=0; i<v.size(); i++)
      cout << v[i];
    cout << "\n";
  } while(prev_permutation(v.begin(), v.end()));

  return 0;
}
```

Aquí se muestra la salida del programa:

```
Todas las permutaciones de ABC con el uso de next_permutation():
ABC
ACB
BAC
BCA
CAB
CBA

Todas las permutaciones de ABC con el uso de prev_permutation():
CBA
CAB
BCA
BAC
ACB
ABC
```

Opciones

Tanto **next_permutation()** como **prev_permutation()** proporcionan una segunda forma que le permite especificar una función de comparación, que determina cuando un elemento es menor que otro. Puede usar esta función para especificar el orden de la secuencia. (En otras palabras, esta función determina el orden de la secuencia.) Aquí se muestran estas formas:

template <class BiIter, class Comp>
 bool next_permutation(BiIter *inicio*, BiIter *final*, Comp *fucomp*)

template <class BiIter, class Comp>
 bool prev_permutation(BiIter *inicio*, BiIter *final*, Comp *fucomp*)

El orden de permutación se basará en *fucomp*. De otra manera, estas funciones trabajarán como sus versiones previamente descritas.

Los algoritmos **next_permutation()** y **prev_permutation()** generan permutaciones en un orden bien definido. En algunas situaciones, tal vez quiera volver aleatoria la generación de permutaciones. Una manera de hacer esto es con el algoritmo **random_shuffle()**. Ordena una secuencia de manera aleatoria. Aquí se muestra una de sus formas:

template <class RandIter> void random_shuffle(RandIter *inicio*, RandIter *final*)

Vuelve aleatorio el rango *inicio* a *final*-1. De acuerdo con el programa anterior de ejemplo, lo siguiente produce una permutación aleatoria de **v**:

```
random_shuffle(v.begin(), v.end());
```

También hay una segunda forma de **random_shuffle()** que le permite especificar un generador de números aleatorios personalizados. Consulte *Invierta, gire y modifique el orden de una secuencia*.

Copie una secuencia de un contenedor a otro

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template <class InIter, class OutIter> OutIter copy(InIter *inicio*, InIter *final*, OutIter *resultado*)

Aunque es conceptualmente simple, uno de los algoritmos STL más importantes es **copy()**, que copia una secuencia. También es importante porque le da una manera de copiar elementos de un contenedor a otro. Más aún, no es necesario que los tipos de contenedor sean iguales. Por ejemplo, empleando **copy()**, puede copiar elementos de un **vector** a una **list**. Por supuesto, lo que hace esto posible es el hecho de que **copy()** (como casi todos los algoritmos de STL) funciona mediante iteradores. Se ha dicho que los iteradores son el pegamento que une la STL. El algoritmo **copy()** ilustra este punto, y en esta solución se muestra cómo ponerlo en acción.

Paso a paso

Para usar **copy()** con el fin de copiar elementos de un tipo de contenedor a otro se requieren estos pasos:

1. Confirme que el contenedor de destino es lo suficientemente grande como para contener los elementos que se copiarán en él.
2. Llame a **copy()** para copiar los elementos, especificando el rango que habrá de copiarse y un iterador al inicio del destino.

Análisis

Aquí se muestra el algoritmo **copy()**:

template <class InIter, class OutIter>
OutIter copy(InIter *inicio*, InIter *final*, OutIter *resultado*)

Este algoritmo copia el rango *inicio* a *final*-1 en la secuencia de destino, empezando en *resultado*. Devuelve un apuntador a uno después del final de la secuencia resultante. He aquí un tema importante: los elementos copiados *no* se agregan al contenedor de destino. En cambio, *sobrescriben* elementos existentes. Por tanto, el contenedor de destino al que señala *resultado* debe ser lo suficientemente grande como para contener los elementos que habrán de copiarse. El algoritmo **copy()** *no* aumentará automáticamente el tamaño del contenedor de destino cuando se copian elementos en él. El algoritmo simplemente supone que el contenedor de destino es lo suficientemente grande.

No es necesario que *resultado* señale al mismo contenedor que *inicio* a *final*, ni que use el mismo contenedor. Esto significa que puede usar **copy()** para copiar el contenido de un tipo de contenedor en otro. La única restricción es que el tipo de elemento del contenedor de destino debe ser compatible con el de origen.

Otro aspecto útil de **copy()** es que puede usarse para desplazar elementos a la izquierda dentro del mismo rango, siempre y cuando el último elemento del rango no se superponga con el rango de destino.

Ejemplo

En el siguiente ejemplo se muestra la manera de usar **copy()** para copiar elementos de una **list** en un **vector**.

```
// Usa copy() para copiar elementos de una lista a un vector.

#include <iostream>
#include <vector>
#include <list>
#include <algorithm>

using namespace std;

template<class T> void mostrar(const char *msj, T cont);

int main()
{
  list<char> lista;

  // Agrega elementos a lista.
  char cad[] = "Los algoritmos act\u00a3n sobre los contenedores";
  for(int i = 0; cad[i]; i++) lista.push_back(cad[i]);

  // Crea un vector que contiene 53 puntos al principio.
  vector<char> v(53, '.');

  mostrar("Contenido de lista:\n", lista);
  mostrar("Contenido de v:\n", v);

  // Copia lista en v.
  copy(lista.begin(), lista.end(), v.begin()+5);

  // Despliega resultado.
  mostrar("Contenido de v tras la copia:\n", v);
  return 0;
}

template<class T> void mostrar(const char *msj, T cont) {
  cout << msj;
  T::iterator itr;
  for(itr=cont.begin(); itr != cont.end(); ++itr)
    cout << *itr;

  cout << "\n\n";
}
```

Aquí se muestra la salida:

```
Contenido de lista:
Los algoritmos actúan sobre los contenedores

Contenido de v:
.....................................................

Contenido de v tras la copia:
.....Los algoritmos actúan sobre los contenedores.....
```

Opciones

La STL proporciona dos variaciones útiles de **copy()**. La primera es **copy_backward()**, que se muestra aquí:

template <class BiIter1, class BiIter2>
  BiIter2 copy_backward(BiIter1 *inicio*, BiIter1 *final*, BiIter2 *resultado*)

Este algoritmo funciona como **copy()**, excepto porque mueve primero elementos del final del rango especificado, y *resultado* debe señalar inicialmente a uno después del principio del rango de destino. Por tanto, puede usarse para desplazar a la derecha elementos dentro del mismo rango, siempre y cuando el primer elemento del rango no se superponga al rango de destino.

La segunda opción de copia es **swap_ranges()**. Intercambia el contenido de un rango con otro. Por tanto, proporciona una copia bidireccional. Se muestra aquí:

template <class ForIter1, class ForIter2>
  ForIter2 swap_ranges(ForIter *inicio1*, ForIter *inicio2*, ForIter2 *inicio2*)

El algoritmo **swap_ranges()** intercambia elementos en el rango *inicio1* a *final1*-1 con elementos en la secuencia que empieza en *inicio2*. Devuelve un apuntador al final de la secuencia especificada por *inicio2*. No deben superponerse los rangos que se intercambian.

Reemplace y elimine elementos en un contenedor

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algortihm>		template <class ForIter, class T> ForIter remove(ForIter *inicio*, ForIter *final*, const T &*val*) template <class ForIter, class T> void replace(ForIter *inicio*, ForIter *final*, const T &*ant*, const T &*nue*)

La STL proporciona funciones que le permiten reemplazar o eliminar elementos. En el núcleo de la funcionalidad están **replace()** y **remove()**. Aunque ambas operaciones pueden realizarse mediante el uso de funciones definidas por el contenedor, en muchos casos, estos algoritmos depuran la tarea. En esta solución se demuestran.

Paso a paso

Para eliminar o reemplazar uno o más elementos en una secuencia se requieren estos pasos:

1. Para eliminar todos los elementos que coinciden con un valor específico, llame a **remove()**, especificando el rango que habrá de eliminarse y el valor que se eliminará.
2. Para reemplazar todas las apariciones de elementos que coinciden con un valor específico, llame a **replace()**, especificando el rango que habrá de modificarse, y el valor que se reemplazará y el que se sustituirá.

Análisis

El algoritmo **remove()** elimina todas las apariciones de un elemento especificado de un rango especificado. Aquí se muestra:

template <class ForIter, class T>
ForIter remove(ForIter *inicio*, ForIter *final*, const T &*val*)

Este algoritmo elimina todos los elementos del rango *inicio* a *final*-1 que son iguales a *val*. Devuelve un iterador al final de los elementos restantes. El orden de los elementos restantes queda sin cambio.

Dentro de un rango especificado, el algoritmo **replace()** reemplaza todas las apariciones de un elemento especificado con otro. Aquí se muestra:

template <class ForIter, class T>
void replace(ForIter *inicio*, ForIter *final*, const T &*ant*, const T &*nue*)

Dentro del rango especificado *inicio* a *final*-1, **replace()** reemplaza elementos que coinciden con el valor *ant* con elementos que tienen el valor *nue*.

NOTA *La clase contenedora **list** proporciona su propia implementación de **remove()** que está optimizada para listas. Por tanto, cuando se eliminan elementos de una **list**, debe usar la función en lugar del algoritmo **remove()**.*

Ejemplo

En el siguiente ejemplo se muestran **remove()** y **replace()**:

```
// Demuestra remove() y replace().

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template<class InIter>
    void mostrar_rango(const char *msj, InIter inicio, InIter final);
```

```
int main()
{
  vector<char> v;
  vector<char>::iterator itr, itr_final;

  // Crea un vector que contiene A B C D E A B C D E.
  for(int i=0; i<5; i++) {
    v.push_back('A'+i);
  }
  for(int i=0; i<5; i++) {
    v.push_back('A'+i);
  }

  mostrar_rango("Contenido original de v:\n", v.begin(), v.end());
  cout << endl;

  // Elimina todas las A.
  itr_final = remove(v.begin(), v.end(), 'A');

  mostrar_rango("v tras eliminar todas las A:\n", v.begin(), itr_final);
  cout << endl;

  // Reemplaza B con dígitos X.
  replace(v.begin(), v.end(), 'B', 'X');

  mostrar_rango("v tras reemplazar B con X:\n", v.begin(), itr_final);
  cout << endl;

  return 0;
}

// Muestra un rango de elementos de un vector<char>.
template<class InIter>
    void mostrar_rango(const char *msj, InIter inicio, InIter final) {
  InIter itr;

  cout << msj;
  for(itr = inicio; itr != final; ++itr)
    cout << *itr << " ";
  cout << endl;
}
```

Aquí se muestra la salida:

```
Contenido original de v:
A B C D E A B C D E

v tras eliminar todas las A:
B C D E B C D E

v tras reemplazar B con X:
X C D E X C D E
```

Opciones

La STL proporciona varias opciones para eliminar y reemplazar elementos. Dos que le resultarán particularmente útiles son **remove_copy()** y **replace_copy()**. Ambas generan una nueva secuencia que contiene el resultado de la operación. Por tanto, la secuencia original queda sin alteración.

Aquí se muestra el prototipo para **remove_copy()**:

template <class InIter, class OutIter, class T>
 OutIter remove_copy(InIter *inicio*, InIter *final*, OutIter *resultado*, const T &*val*)

Copia elementos del rango especificado, eliminando los que sean iguales a *val*. Pone el resultado en la secuencia a la que señala *resultado* y devuelve un iterador a uno después del final del resultado. El rango de destino debe ser lo suficientemente grande para contener el resultado.

El prototipo para **replace_copy()** se muestra a continuación:

template <class InIter, class OutIter, class T>
 OutIter replace_copy(InIter *inicio*, InIter *final*,
 OutIter *resultado*, const T &*ant*, const T &*nue*)

Copia elementos del rango especificado, reemplazando elementos iguales a *ant* con *nue*. Coloca el resultado en la secuencia señalada por *resultado* y devuelve un iterador a uno después del final del resultado. El rango de destino debe tener el tamaño suficiente para contener el resultado.

Hay variaciones de **remove()**, **replace()**, **remove_copy()** y **replace_copy()** que le permiten especificar un predicado unario que determina cuándo debe eliminarse o reemplazarse un elemento. Se les denomina **remove_if()**, **replace_if()**, **remove_copy_if()** y **replace_copy_if()**.

Otro algoritmo que elimina elementos de una secuencia es **unique()**. Elimina elementos duplicados consecutivos de un rango. Tiene las dos formas mostradas aquí:

template <class ForIter>
 ForIter unique(ForIter *inicio*, ForIter *final*)

template <class ForIter, class BinPred>
 ForIter unique(ForIter *inicio*, ForIter *final*, BinPred *funp*)

Se eliminan elementos duplicados consecutivos en el rango especificado. La segunda forma le permite especificar un predicado binario que determina cuando un elemento es igual a otro. **unique()** devuelve un iterador al final del rango resultante. Por ejemplo, suponiendo el programa anterior, si **v** contiene la secuencia AABCCBDE, entonces después de la ejecución de esta instrucción

```
itr_final = unique(v.begin(), v.end());
```

el rango **v.begin()** a **itr_final** contendrá ABCBDE. La STL también proporciona **unique_copy()**, que funciona de la misma manera que **unique()**, excepto que el resultado se coloca en otra secuencia.

Combine dos secuencias ordenadas

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template <class InIter1, class InIter2, class OutIter> OutIter merge(InIter1 *inicio1*, InIter1 *final1*, InIter2 *inicio2*, InIter2 *final2* OutIter *resultado*) template <class BiIter> void inplace_merge(BiIter *inicio*, BiIter *medio*, BiIter *final*)

Hay dos algoritmos de STL que combinan dos secuencias ordenadas: **merge()** e **inplace_merge()**. Para ambos, el resultado es una secuencia ordenada que incluye el contenido de las dos secuencias originales. Como recordará, la mezcla está directamente apoyada por el contenedor **list**. Sin embargo, no es proporcionada por otros contenedores integrados. Por tanto, si quiere combinar secuencias de elementos de cualquier otra cosa diferente de un contenedor **list**, necesitará usar uno de los algoritmos de mezcla.

Hay dos maneras en que puede realizarse una mezcla. En primer lugar, puede almacenarse el resultado en una tercera secuencia. En segundo lugar, si la mezcla incluye dos secuencias del mismo contenedor, entonces el resultado puede almacenarse en el lugar. El primer método es proporcionado por **merge()**, y el segundo por **inplace_merge()**. En esta solución se ilustran ambos.

Paso a paso

Para mezclar dos secuencias, almacenando el resultado en una tercera secuencia, se requieren estos pasos:

1. Asegúrese de que las secuencias que se mezclarán están ordenadas.
2. Llame a **merge()**, pasándola en los rangos que habrán de mezclarse y un iterador al principio del rango de destino que contendrá el resultado.

Para mezclar dos secuencias en el lugar se requieren estos pasos:

1. Asegúrese de que las secuencias que habrán de mezclarse están ordenadas.
2. Llame a **inplace_merge()**, pasándola en los rangos que habrán de mezclarse. El resultado se almacenará en el lugar.

Análisis

El algoritmo **merge()** mezcla dos secuencias ordenadas y almacena el resultado en una tercera secuencia. Tiene dos formas. La usada en esta solución se muestra a continuación:

```
template <class InIter1, class InIter2, class OutIter>
  OutIter merge(InIter1 inicio1, InIter1 final1
                InIter2 inicio2, InIter2 final2
                OutIter resultado)
```

El algoritmo **merge()** mezcla dos secuencias ordenadas, colocando el resultado en una tercera secuencia. Los rangos que habrán de mezclarse están definidos por *inicio1*, *final1* e *inicio2*, *final2*. El resultado se pone en el contenedor señalado por *resultado*. Este contenedor *debe* ser del tamaño suficiente para contener los elementos que se almacenarán en él, porque los elementos mezclados sobrescriben los elementos existentes. El algoritmo **merge()** no inserta nuevos elementos. Se devuelve un iterador a uno después del final de la secuencia resultante.

Es importante comprender que **merge()** no requiere que la secuencia de entrada o la resultante sean del mismo tipo de contenedor. Por ejemplo, puede usar **merge()** para mezclar una secuencia de una instancia de **vector** con una secuencia de una instancia de **deque**, almacenando el resultado en un objeto de **list**. Por tanto, **merge()** ofrece una manera de combinar elementos de contenedores separados.

El algoritmo **inplace_merge()** realiza una mezcla en dos rangos ordenados consecutivos dentro del mismo contenedor, y el resultado reemplaza a los dos rangos originales. Tiene dos formas. Aquí se muestra la usada por esta solución:

template <class BiIter>
 void inplace_merge(BiIter *inicio*, BiIter *medio*, BiIter *final*)

Dentro de una sola secuencia, el algoritmo **inplace_merge()** mezcla el rango de *inicio* a *medio*-1 con el rango de *medio* a *final*-1; ambos rangos deben estar ordenados. Después de la ejecución, la secuencia resultante se ordena y está contenida en el rango *inicio* a *final*-1.

NOTA *La **clase** del contenedor **list** proporciona su propia implementación de **merge()** que está optimizado para listas. Por tanto, cuando se mezclan listas, debe usar esa función en lugar del algoritmo **merge()**.*

Ejemplo

En el siguiente ejemplo se muestran **merge()** e **inplace_merge()** en acción. Se usa **merge()** para mezclar un **vector** con una **deque**. El resultado se almacena en una **list**. Observe que tanto las secuencias de entrada como el resultado están ordenados. Luego se usa **inplace_merge()** para mezclar dos secuencias dentro del mismo vector.

```
// Demuestra merge() e inplace_merge().

#include <iostream>
#include <vector>
#include <deque>
#include <list>
#include <algorithm>

using namespace std;

template<class InIter>
    void mostrar_rango(const char *msj, InIter inicio, InIter final);

int main()
{
  vector<char> v;
  deque<char> dq;
  list<char> resultado(26);
```

```
  list<char>::iterator res_final;

  // Primero, demuestra merge().

  for(int i=0; i < 26; i+=2) v.push_back('A'+i);
  for(int i=0; i < 26; i+=2) dq.push_back('B'+i);

  mostrar_rango("Contenido original de v:\n", v.begin(), v.end());
  cout << endl;

  mostrar_rango("Contenido original de dq:\n", dq.begin(), dq.end());
  cout << endl;

  // Mezcla v con dq.
  res_final = merge(v.begin(), v.end(),
                    dq.begin(), dq.end(),
                    resultado.begin());

  mostrar_rango("Resultado de mezclar v con dq:\n", resultado.begin(), res_final);
  cout << "\n\n";

  // Ahora, demuestra inplace_merge().

  vector<char> v2;
  for(int i=0; i < 26; i+=2) v2.push_back('B'+i);
  for(int i=0; i < 26; i+=2) v2.push_back('A'+i);

  mostrar_rango("Contenido original de v2:\n", v2.begin(), v2.end());
  cout << endl;

  // Mezcla dos rangos de v2.
  inplace_merge(v2.begin(), v2.begin()+13, v2.end());

  mostrar_rango("Contenido de v2 tras mezclar en el lugar:\n", v2.begin(),
  v2.end());

  return 0;
}

// Muestra un rango de elementos.
template<class InIter>
    void mostrar_rango(const char *msj, InIter inicio, InIter final) {

  InIter itr;

  cout << msj;

  for(itr = inicio; itr != final; ++itr)
    cout << *itr << " ";
  cout << endl;
}
```

Aquí se muestra la salida:

```
Contenido original de v:
A C E G I K M O Q S U W Y

Contenido original de dq:
B D F H J L N P R T V X Z

Resultado de mezclar v con dq:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z


Contenido original de v2:
B D F H J L N P R T V X Z A C E G I K M O Q S U W Y

Contenido de v2 tras mezclar en el lugar:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Opciones

Hay una segunda forma de **merge()** que le permite especificar una función de comparación que determina cuando un elemento es menor que otro. Aquí se muestra:

template <class InIter1, class InIter2, class OutIter, class Comp>
 OutIter merge(InIter1 *inicio1*, InIter1 *final1*
  InIter2 *inicio2*, InIter2 *final2*
  OutIter *resultado*, Comp *fucomp*)

Funciona igual que la primera forma, excepto que *fucomp* se usa para comparar dos elementos. Cuando se usa esta manera, la secuencia que se está mezclando también debe ordenarse de acuerdo con *fucomp*.

Hay también una segunda forma de **inplace_merge()** que le permite especificar una función de comparación. Se muestra aquí:

template <class BiIter, class Comp>
 void inplace_merge(BiIter *inicio*, BiIter *medio*, BiIter *final*, Comp *fucomp*)

Funciona como la primera versión, excepto que usa *fucomp* para determinar cuando un elemento es menor que otro. Como es de esperar, las secuencias también deben ordenarse de acuerdo con *fucomp*.

Cree y administre un heap

Componentes clave		
Encabezados	**Clases**	**Funciones**
<algorithm>		template <class RandIter> void make_heap(RandIter *inicio*, RandIter *final*) template <class RandIter> void pop_heap(RandIter *inicio*, RandIter *final*) template <class RandIter> void push_heap(RandIter *inicio*, RandIter *final*) template <class RandIter> void sort_heap(RandIter *inicio*, RandIter *final*)

Un heap, o montón, es una estructura de datos en que el elemento superior (también llamado el primer elemento) es el elemento más grande de la secuencia. Los heaps permiten la inserción y eliminación rápida (en tiempo logarítmico) de un elemento. Son útiles para crear colas de prioridad en que el elemento de mayor prioridad debe estar disponible inmediatamente, pero no se necesita una lista completamente ordenada. La STL proporciona cuatro algoritmos que dan soporte a operaciones con heaps, y en esta solución se demuestra su uso.

Paso a paso

Para crear y administrar un heap, se requieren estos pasos:

1. Para crear un heap, llame a **make_heap()**, especificando el rango de elementos que habrá de crearse en un heap.
2. Para agregar un elemento a un heap, llame a **push_heap()**.
3. Para eliminar un elemento del heap, llame a **pop_heap()**.
4. Para ordenar el heap, llame a **sort_heap()**.

Análisis

Un heap se construye usando el algoritmo **make_heap()**. Tiene dos formas. Aquí se muestra la usada en esta solución:

template <class RandIter>
  void make_heap(RandIter *inicio*, RandIter *final*)

Construye un heap a partir de la secuencia definida por *inicio* a *final*. Cualquier contenedor que da soporte a los iteradores de acceso aleatorio puede usarse para contener un heap. La construcción de un heap ocupa tiempo lineal.

Puede incluir un nuevo elemento en el heap usando **push_heap()**. Tiene dos formas. La usada en esta solución se muesta a continuación:

template <class RandIter>
  void pop_heap(RandIter *inicio*, RandIter *final*)

Coloca el elemento en *final*-1 en el heap definido por *inicio* a *final*-2. En otras palabras, el heap actual termina en *final*-2 y **push_heap()** agrega el elemento en *final*-1. El resultado es un heap que termina en *final*-1. La inclusión de un elemento en un heap consume tiempo logarítmico.

Puede eliminar un elemento usando **pop_heap()**. Tiene dos formas. Aquí se muestra la usada en esta solución:

template <class RandIter>
void push_heap(RandIter *inicio*, RandIter *final*)

El algoritmo **pop_heap()** intercambia los elementos de *inicio* y *final*-1 y luego reconstruye el heap. El heap resultante termina en *final*-2. La eliminación de un elemento de un heap consume tiempo logarítmico.

Puede ordenar un heap de manera ascendente usando **sort_heap()**. Aquí se muestra su prototipo:

template <class RandIter>
void sort_heap(RandIter *inicio*, RandIter *final*)

El algoritmo **sort_heap()** ordena un heap dentro del rango especificado por *inicio* y *final*. El ordenamiento de un heap requiere tiempo proporcional a N log N.

Ejemplo

He aquí un programa que construye un heap, luego agrega y elimina elementos. Termina por ordenar el heap.

```
// Demuestra los algoritmos de heap.

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void mostrar(const char *msj, vector<char> vect);

int main()
{
  vector<char> v;
  int i;

  for(i=0; i<20; i+=2) v.push_back('A'+i);

  mostrar("v antes de construir el heap:\n", v);
  cout << endl;

  // Construye un heap.
  make_heap(v.begin(), v.end());

  mostrar("v tras construir el heap:\n", v);
  cout << endl;

  // Incluye H en el heap.
```

```
  v.push_back('H'); // primero coloca H en el vector
  push_heap(v.begin(), v.end()); // ahora, coloca H en el heap

  mostrar("v tras incluir H en el heap:\n", v);
  cout << endl;

  // Extrae un valor del heap.
  pop_heap(v.begin(), v.end());

  mostrar("v tras extraer un valor del heap:\n", v);
  cout << endl;

  // Ordena el heap
  sort_heap(v.begin(), v.end()-1);
  mostrar("v tras ordenar el heap:\n", v);

  return 0;
}

// Despliega el contenido de un vector<char>.
void mostrar(const char *msj, vector<char> vect) {
  cout << msj;
  for(unsigned i=0; i < vect.size(); ++i)
    cout << vect[i] << " ";
  cout << "\n";
}
```

He aquí la salida del programa:

```
v antes de construir el heap:
A C E G I K M O Q S

v tras construir el heap:
S Q M O I K E A G C

v tras incluir H en el heap:
S Q M O I K E A G C H

v tras extraer un valor del heap:
Q O M H I K E A G C S

v tras ordenar el heap:
A C E G H I K M O Q S
```

Observe el contenido de **v** tras llamar a **pop_heap()**. La S aún está presente, pero ahora se encuentra al final. Como se describió, la eliminación de un elemento de un heap hace que el primer elemento se mueva al final y luego se construye un nuevo heap sobre los elementos restantes (N-1). Por tanto, aunque el elemento eliminado (S, en este caso) permanece en el contenedor, no es parte del heap. Además, observe que la llamada a **sort_heap()** especifica **v.end()-1** como punto final del ordenamiento. Esto se debe a que la S ya no es parte del heap, porque se ha eliminado en el paso anterior.

Opciones

Todas las funciones del heap tienen una segunda forma que le permite especificar una función de comparación que determina cuando un elemento es menor que otro. Aquí se muestran estas versiones:

template <class RandIter, class Comp>
 void make_heap(RandIter *inicio*, RandIter *final*, Comp *fucomp*)

template <class RandIter, class Comp>
 void pop_heap(RandIter *inicio*, RandIter *final*, Comp *fucomp*)

template <class RandIter, class Comp>
 void push_heap(RandIter *inicio*, RandIter *final*, Comp *fucomp*)

template <class RandIter, class Comp>
 void sort_heap(RandIter *inicio*, RandIter *final*, Comp *fucomp*)

En todos los casos, *fucomp* especifica la función de comparación usada para determinar el orden de los elementos.

Aunque los algoritmos de heap son útiles, requieren que usted maneje manualmente el heap. Por fortuna, hay un método más fácil que es aplicable a muchas situaciones: el adaptador de contenedor **priority_queue**, el cual mantiene automáticamente los elementos en el contenedor en orden de prioridad.

Cree un algoritmo

Componentes clave		
Encabezados	**Clases**	**Funciones**
		template<*tipos-iter*, *otros-tipos*> *tipo-ret nombre*(*args-iter*, *otros-args*) template<*tipos-iter*, *otros-tipos*, *tipo_pred*> *tipo-ret nombre*(*args-iter*, *otros-args*, *predicado*)

Aunque la STL proporciona un rico conjunto de algoritmos integrados, también puede crear los propios. Esto es posible porque la STL se diseñó para acomodar extensiones fácilmente. Siempre y cuando siga unas cuantas reglas simples, sus algoritmos serán completamente compatibles con los contenedores de STL y otros elementos. Por tanto, al crear sus propios algoritmos, se expandirá su marco conceptual de STL para cubrir sus necesidades. En esta solución se muestra el proceso.

Paso a paso

Para crear sus propios algoritmos, se requieren estos pasos:

1. Cree una función de plantilla que tome uno o más iteradores como argumentos.
2. Realice todas las operaciones mediante los iteradores pasados a la función.
3. Si se necesita un predicado, inclúyalo en la lista de parámetros para la función, y luego defina el predicado.

Análisis

En general, el proceso de crear un algoritmo es simple. Sólo cree una plantilla de función que opere mediante iteradores que se pasan como argumentos. (Técnicamente, un algoritmo también puede operar mediante referencias, pero casi todo el tiempo deben usarse iteradores.) El tipo de iterador es especificado por un parámetro de plantilla. Por tanto, el prototipo de un algoritmo personalizado tendrá el aspecto de los prototipos de los algoritmos integrados. Tenga en mente un tema importante: el nombre de tipo genérico que le dé a un iterador no tiene efecto en los tipos de iteradores que puede realmente usar cuando llame al algoritmo. Los nombres de tipo de iterador genéricos son simples convenciones que documentan los tipos de iteradores requeridos por el algoritmo. Por tanto, el uso del nombre **BiIter** en una plantilla no impone que sólo puedan usarse iteradores con capacidades bidireccionales. En cambio, son los operadores aplicados al iterador dentro del algoritmo los que determinan cuáles capacidades se requieren. Por ejemplo, si aplica + o – al iterador, entonces sólo pueden usarse iteradores de acceso aleatorio como argumentos.

En principio, un algoritmo personalizado puede devolver cualquier tipo de valor. Por ejemplo, considere la amplia variedad de tipos de devolución encontrados en los algoritmos integrados. **find()** devuelve un iterador, **count()** devuelve un valor entero y **equal()** devuelve un resultado booleano. No obstante lo anterior, he aquí una buena regla a seguir: cuando tenga sentido que su algoritmo devuelva un iterador, debe hacerlo. Esto a menudo hace que su algoritmo sea más versátil porque permite que el resultado de un algoritmo sea usado como entrada de otro. Por supuesto, la naturaleza específica de su algoritmo determinará su tipo de devolución.

Si su algoritmo necesita usar un predicado, incluya un parámetro de plantilla para el predicado. Luego, proporcione el predicado cuando se llame al algoritmo.

Al unir todo, he aquí las principales formas generales de un algoritmo:

template<*tipos-iter*, *otros-tipos*>
  tipo-ret nombre(*args-iter*, *otros-args*)

template<*tipos-iter*, *otros-tipos*, *tipo_pred*>
  tipo-ret nombre(*args-iter*, *otros-args*, *predicado*)

Por supuesto, su aplicación específica determinará el tipo de devolución específico, de argumentos y de predicado.

Como un elemento interesante, en varios de los ejemplos de este capítulo se usa una función llamada **show_range()**. Toma un apuntador a una cadena terminada en un carácter nulo y dos iteradores como argumentos. Luego despliega la cadena seguida por los elementos dentro del rango especificado. Debido a que **show_range()** accede a los elementos mediante iteradores, funciona de manera parecida a un algoritmo. Sin embargo, en opinión del autor no es un algoritmo, en sentido estricto, porque produce salida que está codificada para que se despliegue mediante **cout**. No obstante, muestra la manera en que los iteradores delinean la creación de funciones que pueden aplicarse a contenedores. (Es posible dar salida a información a un flujo mediante un iterador. Consulte *Use los iteradores de flujo* para conocer más detalles.)

Ejemplo

En el siguiente ejemplo se muestra un algoritmo personalizado llamado **disjuntos()**, que compara los elementos en dos rangos. Si no contienen elementos comunes, entonces **disjuntos()** devuelve true. De otra manera, devuelve false.

```
// Esta función es un algoritmo que determina si el contenido
// de dos rangos es disjunto. Es decir, si no contienen
// elementos en común.
template<class InIter>
  bool disjuntos(InIter inicio, InIter final,
                 InIter inicio2, InIter final2) {

  InIter itr;

  for( ; inicio != final; ++inicio)
    for(itr = inicio2; itr != final2; ++itr)
      if(*inicio == *itr) return false;

  return true;
}
```

Como puede ver, todas las operaciones ocurren mediante iteradores. Debido a que los iteradores sólo se mueven en dirección directa y a que recuperan pero no almacenan valores, **disjuntos()** puede llamarse con cualquier tipo de iterador que da soporte a operaciones de entrada.

El siguiente programa pone a **disjuntos()** en acción. Observe que el programa también usa la función **mostrar_rango()**, que despliega los elementos dentro de un rango. Como se mencionó, esta función se usa en varios de los ejemplos de este capítulo y funciona de manera similar a un algoritmo porque opera mediante iteradores.

```
// Este programa demuestra el algoritmo disjuntos().

#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

template<class InIter>
  void mostrar_rango(const char *msj, InIter inicio, InIter final);

template<class InIter>
  bool disjuntos(InIter inicio, InIter final,
                 InIter inicio2, InIter final2);

int main()
{
  list<char> lista1, lista2, lista3;

  for(int i=0; i < 5; i++) lista1.push_back('A'+i);
  for(int i=6; i < 10; i++) lista2.push_back('A'+i);
  for(int i=8; i < 12; i++) lista3.push_back('A'+i);

  mostrar_rango("Contenido de lista1: ", lista1.begin(), lista1.end());
```

```
  mostrar_rango("Contenido de lista2: ", lista2.begin(), lista2.end());
  mostrar_rango("Contenido de lista3: ", lista3.begin(), lista3.end());

   cout << endl;

   // Prueba lista1 y lista2.
  if(disjuntos(lista1.begin(), lista1.end(), lista2.begin(), lista2.end()))
     cout << "lista1 y lista2 son disjuntos\n";
  else cout << "lista1 y lista2 no son disjuntos.\n";

   // Prueba lista2 y lista3.
  if(disjuntos(lista2.begin(), lista2.end(), lista3.begin(), lista3.end()))
     cout << "lista2 y lista3 son disjuntas\n";
  else cout << "lista2 y lista3 no son disjuntas.\n";

  return 0;
}

 // Muestra un rango de elementos.
template<class InIter>
    void mostrar_rango(const char *msj, InIter inicio, InIter final) {

  InIter itr;

  cout << msj;

  for(itr = inicio; itr != final; ++itr)
    cout << *itr << " ";
  cout << endl;
}

// Esta función es un algoritmo que determina si el contenido
// de dos rangos es disjunto. Es decir, si no contienen
// elementos en común.
template<class InIter>
  bool disjuntos(InIter inicio, InIter final,
                 InIter inicio2, InIter final2) {

  InIter itr;

  for( ; inicio != final; ++inicio)
    for(itr = inicio2; itr != final2; ++itr)
      if(*inicio == *itr) return false;

  return true;
}
```

Aquí se muestra la salida:

```
Contenido de lista1: A B C D E
Contenido de lista2: G H I J
Contenido de lista3: I J K L

lista1 y lista2 son disjuntas
lista2 y lista3 no son disjuntas.
```

Ejemplo adicional: use un predicado con un algoritmo personalizado

Es fácil agregar un predicado, como una función de comparación, a un algoritmo. Simplemente especifique un tipo genérico para la función y luego incluya un parámetro de ese tipo en la lista de argumentos. Dentro del algoritmo, llame a la función cuando sea necesario mediante su parámetro. Por ejemplo, he aquí una sobrecarga de **disjuntos()** que le permite especificar un predicado que determina cuando un elemento es igual a otro:

```
// Esta sobrecarga de disjuntos() permite especificar una función
// de compración que determina cuando dos elementos son iguales.
template<class InIter, class Comp>
  bool disjuntos(InIter inicio, InIter final,
                 InIter inicio2, InIter final2, Comp fucomp) {

  InIter itr;

  for( ; inicio != final; ++inicio)
    for(itr = inicio2; itr != final2; ++itr)
      if(fucomp(*inicio, *itr)) return false;

  return true;
}
```

Preste especial atención al parámetro **fucomp**. Puede recibir un apuntador a función o un objeto de función. Luego se utiliza esta función para determinar cuando dos elementos son iguales. En el siguiente programa se demuestra esta versión de **disjuntos()** para ignorar diferencias entre mayúsculas y minúsculas cuando se determina si dos rangos de caracteres son disjuntos. Se utiliza la función de predicado binario **igual_ignoramayus()** para determinar cuando dos caracteres son iguales independientemente de las diferencias entre mayúsculas y minúsculas.

```
// Demuestra una versión de disjuntos() que toma una función de comparación.

#include <iostream>
#include <list>
#include <algorithm>
#include <cctype>

using namespace std;

template<class InIter>
  void mostrar_rango(const char *msj, InIter inicio, InIter final);

template<class InIter>
  bool disjuntos(InIter inicio, InIter final,
                 InIter inicio2, InIter final2);

// Sobrecarga disjuntos() para tomar una función de comparación.
template<class InIter, class Comp>
  bool disjuntos(InIter inicio, InIter final,
                 InIter inicio2, InIter final2, Comp fucomp);

bool igual_ignoramayus(char car1, char car2);
```

```
int main()
{
  list<char> lista1, lista2;

  for(int i=0; i < 5; i++) lista1.push_back('A'+i);
  for(int i=2; i < 7; i++) lista2.push_back('a'+i);

  mostrar_rango("Contenido de lista1: ", lista1.begin(), lista1.end());
  mostrar_rango("Contenido de lista2: ", lista2.begin(), lista2.end());

  cout << endl;

  // Prueba lista1 y lista2.
  cout << "Probando lista1 y lista2 de manera sensible a \n";
  cout << "may\u00a3sculas y min\u00a3sculas.\n";
  if(disjuntos(lista1.begin(), lista1.end(), lista2.begin(), lista2.end()))
     cout << "lista1 y lista2 son disjuntas\n";
  else cout << "lista1 y lista2 no son disjuntas.\n";

  cout << endl;

  // Prueba lista1 y lista2, pero ignora las diferencias entre mayúsculas
  // y minúsculas.
  cout << "Probando lista1 y lista2 e ignorando diferencias entre\n";
  cout << "may\u00a3sculas y min\u00a3sculas.\n";
  if(disjuntos(lista1.begin(), lista1.end(), lista2.begin(), lista2.end(),
              igual_ignoramayus))
     cout << "lista1 y lista2 son disjuntas\n";
  else cout << "lista1 y lista2 no son disjuntas.\n";

  return 0;
}

// Muestra un rango de elementos
template<class InIter>
    void mostrar_rango(const char *msj, InIter inicio, InIter final) {

  InIter itr;

  cout << msj;

  for(itr = inicio; itr != final; ++itr)
    cout << *itr << " ";
  cout << endl;
}

// Esta función es un algoritmo que determina si el contenido
// de dos rangos es disjunto. Es decir, si no contienen
// elementos en común.
template<class InIter>
  bool disjuntos(InIter inicio, InIter final,
                 InIter inicio2, InIter final2) {

  InIter itr;
```

```
  for( ; inicio != final; ++inicio)
    for(itr = inicio2; itr != final2; ++itr)
      if(*inicio == *itr) return false;

  return true;
}

// Esta sobrecarga de disjuntos() permite especificar una función
// de comparación que determina cuando dos elementos son iguales.
template<class InIter, class Comp>
  bool disjuntos(InIter inicio, InIter final,
                 InIter inicio2, InIter final2, Comp fucomp) {

  InIter itr;

  for( ; inicio != final; ++inicio)
    for(itr = inicio2; itr != final2; ++itr)
      if(fucomp(*inicio, *itr)) return false;

  return true;
}

// Esta función devuelve true si car1 y car2 representan la misma
// letra, a pesar de diferencias entre mayúsculas y minúsculas.
bool igual_ignoramayus(char car1, char car2) {
  if(tolower(car1) == tolower(car2)) return true;
  return false;
}
```

Aquí se muestra la salida:

```
Contenido de lista1: A B C D E
Contenido de lista2: c d e f g

Probando lista1 y lista2 de manera sensible a
mayúsculas y minúsculas.
lista1 y lista2 son disjuntas.

Probando lista1 y lista2 e ignorando diferencias entre
mayúsculas y minúsculas.
lista1 y lista2 no son disjuntas.
```

Opciones

Aunque la creación de su algoritmo es muy fácil, como se muestra en los ejemplos anteriores, a menudo no necesitará crearlos. En muchos casos, puede alcanzar el resultado deseado al usar **for_each()** o **transform()** y especificar una función que realiza la operación deseada. En otros casos, tal vez pueda usar las formas predicadas de uno de los algoritmos estándar de STL. Por supuesto, cuando ninguno de estos métodos funciona, es simple crear su propio algoritmo.

Use un objeto de función integrado

Componentes clave		
Encabezados	**Clases**	**Funciones**
<functional>	divides equal_to greater greater_equal less less_equal logical_and logical_not logical_or minus modulus multiplies negate not_equal_to plus	ret-type operator(*list-args*)

En esta solución se muestra cómo usar los objetos de función integrada definidos por la STL. Una revisión general de los objetos de función se presentó casi al principio de este capítulo, pero será útil empezar por resumir los puntos clave:

- Los objetos de función son instancias de clases que definen **operator()**.
- Un objeto de función puede usarse en lugar de un apuntador a función, como cuando se pasa un predicado a un algoritmo.
- Hay dos tipos de función de objetos: unarios y binarios. Un objeto de función unaria requiere un argumento; uno binario requiere dos.
- Los objetos de función ofrecen más flexibilidad y, en algunos casos, pueden ser más eficientes que los apuntadores de función.

La STL proporciona varios objetos de función integrados, que son el tema de esta solución. También es posible crear sus propios objetos de función. Ésta se describe en la siguiente solución.

Paso a paso

Para crear un objeto de función integrada se requieren estos pasos:

1. Cree una instancia del objeto de función deseada. Especifique el tipo de datos sobre los que operará en su argumento de tipo.
2. Pase el objeto creado en el paso 1 como un argumento a cualquier algoritmo que requiera un argumento de función.

Análisis

Todos los objetos de función integrados son clases de plantilla, lo que significa que pueden funcionar sobre cualquier tipo de datos para los cuales está definida su operación asociada. Los objetos de función integrada usan el encabezado **<functional>**.

La STL define varios objetos de función binaria y dos objetos de función unaria. Los segundos son **logical_not** y **negate**. Los objetos de función binaria integrados se muestran a continuación:

plus	minus	multiplies	divides	modulus
equal_to	not_equal_to	greater	greater_equal	less
less_equal	logical_and	logical_or		

Cada objeto de función realiza la acción implicada en su nombre. Por ejemplo, **negate** devuelve la negación de un valor, **less** devuelve true si un valor es menor que otro, y **divides** devuelve el resultado de dividir un valor entre otro.

Los dos objetos de función usados en el ejemplo son **negate** y **multiplies**. He aquí cómo se declaran:

```
template <class T> estruct negate : funcion_unaria<T, T> {
  T operator()(const T & a) const;
};

template <class T> estruct multiplies : funcion_binaria<T, T> {
  T operator()(const T & a, const T & b) const;
};
```

Observe que se declaran usando la palabra clave **struct**. Recuerde que en C++, **struct** crea un tipo de clase. Los otros objetos de función se declaran de una manera similar.

Para usar un objeto de función, primero debe construir uno. Por ejemplo:

```
negate<int>()
```

construye un objeto de **negate** para usar en operandos de tipo **int**, y

```
multiplies<double, double>()
```

construye un objeto de **multiplies** para usar en operandos **double**.

A menudo, una instancia de un objeto de función no se construye hasta que en realidad se pasa a un algoritmo. Por ejemplo, esta instrucción:

```
transform(inicio1, final1, inicio2, negate<double>());
```

construye un objeto de función **negate** y lo pasa a **transform()** en un paso. Con frecuencia, no hay necesidad de construir una instancia independiente.

Ejemplo

En el siguiente ejemplo se demuestra el objeto de función unaria **negate** y el binario **multiplies**. La misma técnica se aplica a cualquier objeto de función integrado.

NOTA *Otro ejemplo que utiliza un objeto de función integrado se encuentra en* Ordene un contenedor. *Usa el objeto de función* ***greater*** *para ordenar un contenedor en orden inverso.*

```
// Demuestra los objetos de función negate y multiplies.

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

template<class T> void mostrar(const char *msj, T cont);

int main()
{
  vector<int> v, v2, resultado(10);

  for(unsigned i=0; i < 10; ++i) v.push_back(i);
  for(unsigned i=0; i < 10; ++i) v2.push_back(i);

  mostrar("Contenido de v:\n", v);
  mostrar("Contenido de v2:\n", v2);
  cout << endl;

  // Multiplica v y v2.
  transform(v.begin(), v.end(), v2.begin(), resultado.begin(),
            multiplies<int>());

  mostrar("Resultado de multiplicar los elementos de v con los de v2:\n", resultado);
  cout << endl;

  // Luego, niega el contenido de resultado.
  transform(v.begin(), v.end(), v.begin(), negate<int>());

  mostrar("Tras negar v:\n", v);

  return 0;
}

// Despliega el contenido de un contenedor.
template<class T> void mostrar(const char *msj, T cont) {
  cout << msj;

  T::iterator itr;
  for(itr=cont.begin(); itr != cont.end(); ++itr)
    cout << *itr << " ";

  cout << "\n";
}
```

Aquí se muestra la salida:

```
Contenido de v:
0 1 2 3 4 5 6 7 8 9
Contenido de v2:
0 1 2 3 4 5 6 7 8 9

Resultado de multiplicar los elementos de v con los de v2:
0 1 4 9 16 25 36 49 64 81

Tras negar v:
0 -1 -2 -3 -4 -5 -6 -7 -8 -9
```

Opciones

Como regla general, si un objeto de función integrada manejará la situación, debe usarla. En los casos en que no, puede crear su propio objeto de función, como se describe en la siguiente solución. Otra opción consiste en pasar un apuntador a una función estándar. Por ejemplo, dado un contenedor que contiene una secuencia de caracteres, puede pasar la función **islower()** a **remove_if()** para eliminar todas las minúsculas.

Un objeto de función puede tener un límite de valor mediante el uso de un adhesivo. Consulte *Use un adhesivo* para conocer detalles.

Cree un objeto de función personalizado

Componentes clave		
Encabezados	**Estructuras**	**Funciones y Typedefs**
<functional>	binary_function	argument_type result_type
<functional>	unary_function	first_argument_type second_argument_type result_type
		result_type operator(argument_type *arg*) result_type operator(first_argument_type *arg1*, second_argument_type *arg2*)

Uno de los componentes clave de la STL es el objeto de función. Como se explicó en *Revisión general de objetos de función*, un objeto de función es una instancia de una clase que implementa **operator()**. Por tanto, cuando se ejecuta en el objeto la función que llama al operador, que es (), se ejecuta **operator()**. Un objeto de función puede pasarse a cualquier algoritmo que requiera un apuntador a función. Por tanto, puede usarse un objeto de función como predicado. Hay varios objetos de función integrados, como **less**, y su uso se describe en la solución anterior. También puede crear sus propios objetos de función. En esta solución se muestra el proceso.

Antes de empezar, vale la pena mencionar algunas palabras acerca de la razón por la que podría crear sus propios objetos de función. A primera vista, parecería que los objetos de función requieren un poco más de trabajo que simplemente usar apuntadores de función pero que no ofrecen ventajas. Éste no es el caso. Los objetos de función expanden el alcance y el poder de la STL de tres maneras.

En primer lugar, un objeto de función puede proporcionar un mecanismo más eficiente para el paso de funciones a algoritmos. Por ejemplo, es posible para el compilador poner en línea un objeto de función. En segundo lugar, con el uso de un objeto de función puede simplificarse y estructurarse mejor la implementación de operaciones complicadas, porque la clase que define un objeto de función puede contener valores y proporcionar capacidades adicionales. En tercer lugar, un objeto de función define un tipo de nombre. Una función no. Esto permite que objetos de función se especifiquen como argumentos de tipo de plantilla. Por tanto, aunque no hay nada equivocado con el uso de apuntadores a función donde sea aplicable, los objetos de función ofrecen una opción poderosa.

Paso a paso

Para crear un objeto de función se requieren estos pasos:

1. Cree una clase que implemente **operator()**.
2. Para la mayor flexibilidad, haga que la clase del paso 1 herede la estructura **unary_function** o **binary_function**, dependiendo de si está creando un objeto de función binaria o unaria. Éstos definen los nombres de tipo estándar para el archivo o los argumentos de la función y el tipo que se devuelve.
3. Cuando se implemente la clase, evite crear efectos colaterales.

Análisis

Para crear un objeto de función, defina una clase que sobrecargue la función **operator()** y luego cree una instancia de esa clase. Esta instancia puede pasarse a un algoritmo, que luego puede llamar a la función **operator()** mediante la instancia.

Hay dos tipos de objetos de función: unario y binario. Un objeto de función unaria implementa **operator()** de modo que toma un argumento. Para un objeto de función binaria, **operator()** toma dos argumentos. Tal como se usan con algoritmos de STL, cada argumento recibe un elemento del rango o los rangos en que está operando el algoritmo. Por tanto, el tipo de argumento debe ser compatible con el tipo de elemento que se le pasa.

Todos los objetos de función de STL integrados son clases de plantilla. Sus objetos de función también pueden definirse como clases de plantilla, pero no es obligatorio. En ocasiones, un objeto de función personalizado sirve a un propósito específico y una versión de plantilla no es útil.

Con el fin de obtener la mayor flexibilidad para su objeto de función, su clase debe heredar una de estas estructuras definidas por la STL:

```
template <class Argument, class Result, estruct unary_function {
  typedef Argument argument_type;
  typedef Result result_type;
};
```

```
template <class Argument1, class Argument2, class Result>
struct binary_function {
  typedef Argument1 first_argument_type;
  typedef Argument2 second_argument_type;
  typedef Result result_type;
};
```

Una clase que crea un objeto de función unaria hereda **unary_function**. Una clase que crea uno binario hereda **binary_function**. Tanto **unary_function** como **binary_function** se declaran en el encabezado **<functional>**. En general, deben heredarse como públicas, que es la opción predeterminada para estructuras.

Las estructuras **unary_function** y **binary_function** proporcionan definiciones para el tipo o los tipos de argumentos y el tipo de devolución del objeto de función. Estos nombres se usan con algunos adaptadores y pueden ser útiles en otros casos. Por tanto, debe usar estos nombres en su objeto de función. En otras palabras, debe usar **result_type** como tipo de devolución para **operator()**. Debe usar **argument_type** como tipo de argumento para **operator()** en un objeto de función unaria y usar **first_argument_type** y **second_argument_type** como tipos de los argumentos para un objeto de función binaria. Por tanto, las formas generales de **operator()** tienen este aspecto:

result_type operator(argument_type *arg*)

result_type operator(first_argument_type *arg1*, second_argument_type *arg2*)

Un objeto de función no debe crear efectos colaterales. En otras palabras, no debe realizar acciones no relacionadas con su objetivo. Por ejemplo, un objeto de función cuyo propósito es comparar dos elementos en busca de igualdad no debe modificar uno de los elementos en el proceso.

Ejemplo

En el siguiente ejemplo se muestran casos de objetos de función unaria y binarios. Se vuelve a trabajar el programa de ejemplo de la solución *Use **transform()** para cambiar una secuencia*. En esa versión, los apuntadores de función se pasan al algoritmo **transform()**. Las funciones calculan el recíproco de un valor y el punto medio entre dos valores. Esta versión del programa usa objetos de función en lugar de apuntadores de función. Crea una clase de objeto de función unaria llamado **recíproco** que calcula el recíproco de un valor. Crea una clase de objeto de función binaria llamado **puntomedio** que calcula el punto medio entre dos valores.

```
// Demuestra los objetos de función unaria y binarios.
//
// En este programa se vuelve a trabajar el ejemplo de la
// solución "Use transform() para cambiar una secuencia". En ese
// programa se usaron apuntadores a función en llamadas a transform().
// En esta versión se usan objetos de función.

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
```

```
using namespace std;

// Un objeto de función que calcula un recíproco.
class reciproco : unary_function<double, double> {
public:
  result_type suma;

result_type operator()(argument_type val) {
    if(val == 0.0) return 0.0;
    return 1.0 / val; // devuelve el recíproco
  }
};

// Un objeto de función que encuentra el punto medio entre
// dos valores.
class puntomedio : binary_function<int, int, double> {
public:
  result_type operator()(first_argument_type a, second_argument_type b) {
    return((a-b) / 2) + b;
  }
};

template<class T> void mostrar(const char *msj, vector<T> vect);

int main()
{
  int i;

  vector<double> v;

  // Pone valores en v.
  for(i=1; i < 10; ++i) v.push_back((double)i);

  mostrar("Contenido inicial de v:\n", v);
  cout << endl;

  // Primero, demuestra un objeto de función unaria.

  // Transforma v al aplicar el objeto de función recíproco.
  // Pone de nuevo el resultado en v.
  cout << "Usa un objeto de funci\u00a2n unario en llamadas a transform() para\n";
  cout << "calcular rec\u00a1procos para v y almacenar de nuevo el resultado en
  v.\n";
  transform(v.begin(), v.end(), v.begin(), reciproco());

  mostrar("Contenido transformado de v:\n", v);
  cout << endl;

  // Transforma v por segunda vez, poniendo el resultado en una nueva secuencia.
  cout << "Usa un objeto de funci\u00a2n unario para transformar v de nuevo.\n";
  cout << "Esta vez se almacenan los resultados en v2.\n";
  vector<double> v2(10);
  transform(v.begin(), v.end(), v2.begin(), reciproco());
```

```
   mostrar("He aqu\u00a1 v2:\n", v2);
   cout << endl;

  vector<int> v3, v4, v5(10);
  for(i = 0; i < 10; ++i) v3.push_back(i);
  for(i = 10; i < 20; ++i) if(i%2) v4.push_back(i); else v4.push_back(-i);

  mostrar("Contenido de v3:\n", v3);
  mostrar("Contenido de v4:\n", v4);
  cout << endl;

  // Ahora, demuestra un objeto de función binaria.
  cout << "Ahora, usa un objeto de funci\u00a2n binario para encontrar los puntos
  medios\n";
  cout << "entre elementos en v3 y v4 y almacena los resultados en v5.\n";
  transform(v3.begin(), v3.end(), v4.begin(), v5.begin(), puntomedio());

  mostrar("Contenido de v5:\n", v5);

  return 0;
}

// Despliega el contenido de un vector<int>.
template<class T> void mostrar(const char *msj, vector<T> vect) {
  cout << msj;
  for(unsigned i=0; i < vect.size(); ++i)
    cout << vect[i] << " ";
  cout << "\n";
}
```

Aquí se muestra la salida:

```
Contenido inicial de v:
1 2 3 4 5 6 7 8 9

Usa un objeto de función unaria en llamadas a transform() para
calcular recíprocos para v y almacenar de nuevo el resultado en v.
Contenido transformado de v:
1 0.5 0.333333 0.25 0.2 0.166667 0.142857 0.125 0.111111

Usa un objeto de función unaria para transformar v de nuevo.
Esta vez se almacenan los resultados en v2.
He aquí v2:
1 2 3 4 5 6 7 8 9 0

Contenido de v3:
0 1 2 3 4 5 6 7 8 9
Contenido de v4:
-10 11 -12 13 -14 15 -16 17 -18 19

Ahora, usa un objeto de función binaria para encontrar los puntos medios
entre elementos en v3 y v4 y almacenar los resultados en v5.
Contenido de v5:
-5 6 -5 8 -5 10 -5 12 -5 14
```

Ejemplo adicional: use un objeto de función para mantener información de estado

Aunque en el ejemplo anterior se demuestra cómo crear dos objetos de función diferentes, no muestra la capacidad real de los objetos de función. Por ejemplo, éstos pueden usarse con adhesivos y negadores, y esto se describe en *Use un adhesivo* y *Use un negador*. Otra característica importante de los objetos de función es su capacidad de mantener información de estado. Es posible para la clase que defina un objeto de función para incluir variables de instancia que almacene información acerca del uso del objeto de función, como el resultado de algún cálculo. Esto puede ser útil en varios contextos. Por ejemplo, una variable podría mantener el éxito o la falla de una operación. La capacidad de mantener información de estado expande en gran medida los tipos de problemas a los que puede aplicarse un objeto de función.

En el siguiente ejemplo se demuestra la capacidad de un objeto de función para almacenar información de estado al retrabajar una función de sumatoria usada en el ejemplo de **for_each()** mostrado en *Recorra en ciclo un contenedor con **for_each()***. En ese ejemplo, un apuntador a una función llamada **sumatoria()** se pasaba a **for_each()**. Esta función construía un total constante de los valores en el rango sobre el que operaba **for_each()**. La función **sumatoria()** usaba una variable estática para contener la suma actual. Cada vez que se llamaba a la función, el valor pasado a la función se agregaba al total constante y se devolvía éste (es decir, la sumatoria actual). Aunque este método funcionaba, es poco elegante. Un método mucho mejor consiste en convertir **sumatoria()** en un objeto de función en que el total constante se mantiene en una variable de instancia. No sólo permite que se obtenga la sumatoria sin una llamada a función, también permite que el total se restablezca.

He aquí una manera de crear una clase de objeto de función de sumatoria:

```
// Un objeto de función que calcula una sumatoria entera.
class sumatoria : unary_function<int, void> {
public:
  argument_type suma;

  sumatoria() { suma = 0; }

  // Agregue al total constante y devuelve una
  // referencia al objeto que invoca.
  result_type operator()(argument_type i) {
    suma += i;
  }
};
```

Observe que el total constante se mantiene en un campo llamado **suma** dentro de la clase **sumatoria**. Esto permite que la sumatoria se obtenga del objeto, en lugar de tener que invocar una función. Para restablecer el objeto, simplemente asigna cero a **suma**.

En el siguiente programa se vuelve a trabajar el ejemplo de **for_each()** para que use el objeto de función **sumatoria**:

```
// Usa un objeto de función con for_each().

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
```

```
using namespace std;

// Un objeto de función que calcula una sumatoria entera.
class sumatoria : unary_function<int, void> {
public:
  argument_type suma;

  sumatoria() { suma = 0; }

  // Agregue al total constante y devuelve una
  // referencia al objeto que invoca.
  result_type operator()(argument_type i) {
    suma += i;
  }
};

int main()
{
  vector<int> v;

  for(int i=1; i < 11; i++) v.push_back(i);

  cout << "Contenido de v: ";
  for(unsigned i=0; i < v.size(); ++i)
    cout << v[i] << " ";
  cout << "\n";

  // Declara un objeto de función que recibe el objeto
  // devuelto por for_each().
  sumatoria s;

  // Esto llama a for_each() con un objeto de función, en lugar de
  // un apuntador a función. El objeto de función devuelto por
  // for_each() puede usarse para obtener el total de la sumatoria.
  s = for_each(v.begin(), v.end(), sumatoria());
  cout << "Sumatoria de v: " << s.suma << endl;

  // Cambia el valor de v[4] y vuelve a calcular la sumatoria.
  // Debido a que se crea un nuevo objeto de función, la
  // sumatoria empieza una vez más en cero.
  cout << "Estableciendo v[4] en 99\n";
  v[4]= 99;
  s = for_each(v.begin(), v.end(), sumatoria());
  cout << "La sumatoria de v es ahora: " << s.suma;

  return 0;
}
```

Observe cómo se llama a **for_each()**:

```
s = for_each(v.begin(), v.end(), sumatoria());
```

Se pasa una nueva instancia de **sumatoria**. Este objeto de función se usa mediante esta invocación de **for_each()**. Recuerde que el algoritmo **for_each()** devuelve el objeto de función que se pasa. En tal caso, este objeto se asigna a **s**, que es un objeto de **sumatoria**. Esto significa que **s** contendrá la sumatoria. Este valor se obtiene de **s.suma**.

Opciones

Cuando se usa un objeto de función, tiene la opción de unir un valor a él. Este procedimiento se describe en *Use un adhesivo* y *Use un negador*.

En algunos casos, puede usar un objeto de función integrada, en lugar de uno personalizado. Por ejemplo, si quiere determinar si un valor es mayor que otro, puede usar el objeto de función **greater**. Consulte *Use un objeto de función integrada* para conocer más detalles.

Aunque los objetos de función son más poderosos que los apuntadores de función, no hay nada equivocado en usar uno de estos últimos en situaciones para las que es apropiado. Por ejemplo, si un vector contiene caracteres, entonces es adecuado pasar un apuntador a la función estándar **tolower()** para que **transform()** convierta letras en minúsculas. En este caso, se obtendrían pocos beneficios, si acaso, en la creación de toda una clase para manejar esta operación.

Use un adhesivo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<functional>		template <class Op, class T> binder1st<Op> bind1st(const Op &*obj_fun_bina*, const T &*valor*) template <class Op, class T> binder2nd<Op> bind2nd(const Op &*obj_fun_bina*, const T &*valor*)

En la solución se muestra cómo unir un valor a un objeto de función. Recuerde que un objeto de función binaria toma dos parámetros. Por lo general, estos parámetros reciben valores del rango o los rangos en que está operando el objeto. Por ejemplo, cuando se ordena, la función de comparación binaria recibe pares de elementos del rango que se está ordenando. Aunque el comportamiento predeterminado de un objeto de función binaria es muy útil, hay ocasiones en que querrá modificarlo. Para comprender por qué, tome en consideración lo siguiente.

Suponga que quiere eliminar todos los elementos de una secuencia que son mayores que algún valor, como 10. Su primera idea es, muy naturalmente, usar el objeto de función **greater**. Sin embargo, como opción predeterminada, **greater** recibe ambos valores del rango en que está operando. Por tanto, por sí mismo, no hay manera de hacer que compare elementos de una secuencia con el valor 10. Para usar **greater** con este propósito, necesita alguna manera de *unir* el valor 10 a su operando del lado derecho. Es decir, necesita alguna manera de hacer que **gretaer** realice la siguiente comparación:

val > 10

donde *val* es un elemento de una secuencia. Por fortuna, la STL proporciona un mecanismo, llamado *adhesivos*, que realiza esto. Un adhesivo vincula un valor a uno de los argumentos de un objeto de función binaria. La salida de un adhesivo es un objeto de función unaria, que puede usarse en cualquier lugar en que puede utilizarse cualquier otro objeto de función unaria.

Hay dos adhesivos definidos por la STL: **binds1st()** y **bind2nd()**. En esta solución se muestra su uso.

Paso a paso

Para usar un adhesivo para unir un valor a un objeto de función se requieren estos pasos:

1. Para unir un valor al primer argumento de un objeto de función binaria, llame a **bind1st()**.
2. Para unir un valor al segundo argumento de un objeto de función binaria, llame a **bind2nd()**.
3. Use el resultado del adhesivo en cualquier lugar en que se requiere un predicado unario.

Análisis

Aquí se muestran los prototipos para **binds1st()** y **bind2nd()**:

template <class Op, class T>
 binder1st<Op> bind1st(const Op &*obj_fun_bina*, const T &*valor*)

template <class Op, class T>
 binder2nd<Op> bind2nd(const Op &*obj_fun_bina*, const T &*valor*)

Aquí, *obj_fun_bina* especifica el objeto de función binaria al que se unirá *valor*. **bind1st()** devuelve un objeto de función unaria (encapsulado como un objeto de **binder1st**), que tiene el operando *obj_fun_bina* del lado izquierdo del operando unido a *value*. **bind2nd()** devuelve un objeto de función unaria (encapsulado en un objeto de **binder2nd**) que tiene al operando del lado derecho unido a *valor*. Por ejemplo,

```
bind1st(less<double>, 0.01)
```

une el valor 0.01 al primer argumento (del lado izquierdo) del objeto de función **less**, y

```
bind2nd(less<double>, 0.01)
```

une el valor al segundo argumento (del lado derecho). De los dos, **bind2nd()** es el de uso más común.

Las clases **binder1st** y **binder2nd** representan los objetos de función unaria devueltos por los adhesivos. También se declaran en **<functional>**. Por lo general, no usará directamente la clase **binder1st** o **binder2nd**. En cambio, por lo común pasará la salida de un adhesivo directamente a un algoritmo. Por tanto, **binder1st** y **binder2nd** no se describirán más aquí.

Debido a que un adhesivo convierte un objeto de función binaria en uno unario, el resultado de un adhesivo puede pasarse a cualquier algoritmo que requiere un predicado unario. Por ejemplo, esto pasa un objeto de función unaria a **find_if()**:

```
find_if(v.begin(), v.end(), bind2nd(less<int>, 19))
```

Esto causa que **find_if()** devuelva un iterador al primer valor en **v** que es menor que 19.

Ejemplo

En el siguiente programa se demuestra **bind2nd()**. Utiliza el algoritmo **remove_if()** para eliminar elementos de una secuencia basada en la salida de un predicado. Recuerde que tiene el prototipo:

```
template <class ForIter, class UnPred>
  ForIter remove_if(ForIter inicio, ForIter final, Unpred funp)
```

El algoritmo elimina elementos de la secuencia definida por *inicio* y *final* para el que el predicado unario definido por *funp* es true. El algoritmo devuelve un apuntador al nuevo final de la secuencia, que refleja la eliminación de los elementos.

En el siguiente programa se eliminan todos los valores de una secuencia que es mayor que el valor de 10. Debido a que el predicado requerido por **remove_if** es unario, no podemos simplemente usar el objeto de función **greater** como tal, porque **greater** es un objeto de función binaria. En cambio, debemos unir el valor 10 al segundo argumento de **greater** usando el adhesivo **bind2nd()**.

```
// Demuestra bind2nd().

#include <iostream>
#include <list>
#include <functional>
#include <algorithm>

using namespace std;

template<class InIter>
  void mostrar_rango(const char *msj, InIter inicio, InIter final);

int main()
{
  list<int> lista;
  list<int>::iterator itr_inv;

  for(unsigned i=1; i < 20; ++i) lista.push_back(i);

  mostrar_rango("Secuencia original:\n", lista.begin(), lista.end());
  cout << endl;

  // Usa bind2nd() para crear un objeto de función unaria
  // que devolverá true cuando un valor es mayor que 10.
  // Esto lo usa remove_if() para eliminar todos los
  // elementos de lista que son mayores de 10.
  itr_inv = remove_if(lista.begin(), lista.end(),
                      bind2nd(greater<int>(), 10));
```

```
  mostrar_rango("Secuencia resultante:\n", lista.begin(), itr_inv);

  return 0;
}

// Muestra un rango de elementos.
template<class InIter>
    void mostrar_rango(const char *msj, InIter inicio, InIter final) {

  InIter itr;

  cout << msj;

  for(itr = inicio; itr != final; ++itr)
    cout << *itr << " ";
  cout << endl;
}
```

La salida producida por el programa se muestra aquí.

```
Secuencia original:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Secuencia resultante:
1 2 3 4 5 6 7 8 9 10
```

Como lo muestra la salida, la secuencia resultante contiene los elementos del 1 al 10. Los elementos mayores que 10 se han eliminado. He aquí cómo funciona. Cuando se ejecuta **remove_if()**, el objeto de función binaria **greater** recibe un elemento de **lista** en su primer parámetro y el valor 10 en su segundo, porque el segundo parámetro está unido a 10 empleando **bind2nd()**. Por tanto, para cada elemento de la secuencia, la comparación

elemento > 10

se evaluó. Cuando es true, se elimina el elemento.

Opciones

Aunque **bind2nd()** suele ser el adhesivo más usado de los dos, **bind1st()** está disponible como opción. Como se explicó, el adhesivo **bind1st()** une un valor al primer parámetro. Para ver los efectos, trate de sustituir esta línea en el programa anterior:

```
endp = remove_if(lista.begin(), lista.end(), bind1st(greater<int>(), 10));
```

Esto causa que los elementos de la secuencia se pasen al segundo parámetro de **greater**, con el valor 10 unido al primer parámetro. Por tanto, para cada elemento de la secuencia, se realiza la siguiente comparación:

10 > elemento

Esto causa que **greater** devuelva true para elementos que son menores de 10. Aquí se muestra la salida producida después de que haya sustituido **bind1st()**.

```
Secuencia original:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Secuencia resultante:
10 11 12 13 14 15 16 17 18 19
```

Como puede ver, se han eliminado los elementos que son menores de 10.

Aunque válido, al autor le disgusta el uso de **bind1st()** como se acaba de mostrar porque parece ir en contra de la intuición. Si quiere eliminar elementos que son menores de 10, sería mejor usar esta instrucción:

```
endp = remove_if(lista.begin(), lista.end(), bind2nd(less<int>(), 10));
```

Aquí, se usa el objeto de función **less** y los resultados reflejan lo que se esperaría que ocurra normalmente cuando se emplea **less**. Con el empleo de **bind1st()** y la inversión de la comparación se logran los mismos resultados, pero agrega un poco de confusión sin razón alguna.

Use un negador

Componentes clave		
Encabezados	**Clases**	**Funciones**
<functional>		template <class Pred> unary_negate<Pred> not1(const Pred &*pred_unario*) template <class Pred> unary_negate<Pred> not2(const Pred &*pred_binario*)

Hay un objeto relacionado con un adhesivo, llamado *negador*. Los negadores son **not1()** y **not2()**. Devuelven la negación (es decir, el complemento) de cualquier predicado que modifican. Los negadores delinean la STL porque le permiten adaptar eficientemente un predicado para producir el resultado opuesto, con lo que se evita la necesidad de crear un segundo predicado. En esta solución se demuestra su uso.

Paso a paso

Para usar un negador se requieren estos pasos:

1. Para negar un predicado unario, use **not1()**.
2. Para negar un predicado binario, use **not2()**.

Análisis

Los negadores son **not1()** y **not2()**. Tienen estos prototipos:

template <class Pred> unary_negate<Pred>
 not1(const Pred &*pred_unario*)

template <class Pred> unary_negate<Pred>
 not2(const Pred &*pred_binario*)

El negador **not1()** es para usarse con predicados unarios, y el predicado para negar se pasa en *pred_unario*. Para negar predicados binarios, use **not2()**, pasando el predicado binario en *pred_binario*. El resultado de ambos negadores es un predicado que devuelve la negación del predicado original representado como un objeto de **unary_negate** o **binary_negate**.

Por lo general, no interactuará de manera directa con la clase **unary_negate** o **binary_negate** y no se describen más ampliamente aquí. En cambio, la salida de **not1()** o **not2()** suele pasarse de modo directo a un algoritmo. Por ejemplo, esta instrucción elimina elementos de un contenedor si no son iguales a 'A':

```
remove_if(v.begin(), v.end(), not1(bind2nd(equal_to<char>(), 'A'))):
```

Aunque **equal_to** es un objeto de función binaria, el adhesivo **bind2nd()** lo convierte en un objeto unario. Por esto es por lo que se usa **not1()** en lugar de **not2()**.

Ejemplo

En el siguiente ejemplo se demuestran **not1()** y **not2()**. En primer lugar, muestra un modo de ordenar una secuencia de manera descendente empleando la negación del objeto de función **less** para determinar el orden. Luego se usa **not1()** para eliminar todos los elementos que no son iguales a H.

```
// Demuestra not1() y not2().

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

template<class InIter>
  void mostrar_rango(const char *msj, InIter inicio, InIter final);

int main()
{
  vector<char> v;

  for(int i=0; i < 26; i++) v.push_back('A'+i);

  mostrar_rango("Orden original de v:\n", v.begin(), v.end());
  cout << endl;

  // Usa not2() para invertir el orden de v.
  sort(v.begin(), v.end(), not2(less<char>()));
```

```
  mostrar_rango("Tras ordenar v empleando not2(less<char>()):\n",
                v.begin(), v.end());
  cout << endl;

  // Usa not1() para eliminar todos los elementos que no son iguales a H.
  vector<char>::iterator res_final;
  res_final = remove_if(v.begin(), v.end(),
                        not1(bind2nd(equal_to<char>(), 'H')));

  mostrar_rango("v tras eliminar elementos no iguales a H:\n",
                v.begin(), res_final);

  return 0;
}

// Muestra un rango de elementos.
template<class InIter>
    void mostrar_rango(const char *msj, InIter inicio, InIter final) {
  InIter itr;

  cout << msj;
  for(itr = inicio; itr != final; ++itr)
    cout << *itr << " ";
  cout << endl;
}
```

Produce la siguiente salida:

```
Orden original de v:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Tras ordenar v empleando not2(less<char>()):
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

v tras eliminar elementos no iguales a H:
H
```

Opciones

Aunque puede ser muy útil negar la salida de un predicado y puede mejorar el manejo de muchas situaciones, tal vez no siempre sea la mejor opción. En ocasiones, querrá crear un predicado separado que realice la negación. Por ejemplo, considere un caso en que puede realizarse la negación de alguna operación de manera más eficiente al calcular directamente el resultado negativo, en lugar de revertir la salida del resultado afirmativo. En esta situación, la creación de un predicado aparte es más eficiente que calcular primero el resultado y luego negarlo. En esencia, podría encontrar un caso en que sea más rápido el cálculo de la negación que del resultado afirmativo. En esta situación, no tiene sentido calcular primero la opción afirmativa y luego negarla.

Use el adaptador de apuntador a función

Componentes clave		
Encabezados	**Clases**	**Funciones**
<functional>	pointer_to_unary_function	Result operator()(Arg *arg*) const;
<functional>	pointer_to_binary_function	Result operator()(Arg *arg1*, Arg2 *arg2*) const;
<functional>		template <class Arg, class Result> pointer_to_unary_function<Arg, Result> ptr_fun(Result (**func*)(Arg)) template <class Arg1, class Arg2, class Result> pointer_to_binary_function<Arg1, Arg2, Result> ptr_fun(Result (**func*)(Arg1, Arg2))

El encabezado **<functional>** define varias clases, llamadas *adaptadores a función*, que le permiten adaptar un apuntador a función a una forma que puede usarse con diversos componentes de STL. Varios de estos adaptadores están diseñados para situaciones más allá del alcance de este libro, pero uno resulta especialmente interesante porque resuelve un problema muy común: permitir un apuntador a función que habrá de usarse con un adhesivo o un negador.

Como se ha mostrado en las soluciones anteriores, es posible pasar un apuntador a una función (en lugar de pasar un objeto de función) como un predicado a un algoritmo. Siempre y cuando la función realice la operación deseada, no hay problema en hacer esto. Sin embargo, si quiere unir un valor o usar un negador con esa función, entonces ocurrirán problemas porque no es posible aplicar directamente estos modificadores a apuntadores a función. Para permitir que se usen funciones con adhesivos y negadores, necesitará usar los adaptadores de apuntador a función.

Paso a paso

Para adaptar un apuntador a función en un objeto de función se requieren estos pasos:

1. Para crear un objeto de función a partir de una función unaria, llame a **ptr_fun()**, pasándolo en un apuntador a la función unaria. El resultado es un objeto de función unaria.
2. Para crear un objeto de función a partir de una función binaria, llame a **ptr_fun()**, pasándolo en un apuntador a la función binaria. El resultado es un objeto de función binaria.

Análisis

El adaptador de apuntador a función es **ptr_fun()**. Aquí se muestran ambas formas:

```
template <class Arg, class Result>
  pointer_to_unary_function<Arg, Result>
    ptr_fun(Result (*func)(Arg))

template <class Arg1, class Arg2, class Result>
  pointer_to_binary_function<Arg1, Arg2, Result>
    ptr_fun(Result (*func)(Arg1, Arg2))
```

Devuelve un objeto de tipo **pointer_to_unary** o uno de tipo **pointer_to_binary_function**. Aquí se muestran estas clases:

```
template <class Arg, class Result>
class pointer_to_unary_function:
  public unary_function<Arg, Result>
{
public:
  explicit pointer_to_unary_function(Result (*func)(Arg));
  Result operator() (Arg arg) const;
};

template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function:
  public binary_function<Arg1, Arg2, Result>
{
public:
  explicit pointer_to_binary_function(
             Result (*func)(Arg1, Arg2));
  Result operator() (Arg1 arg1, Arg2 arg2) const;
};
```

Por lo general, no interactuará con estas clases de manera directa. Su principal propósito es construir un objeto de función que encapsula *func*. Para **pointer_to_unary_function**, **operator()** devuelve

func(*arg*)

Y para **pointer_to_binary_function**, **operator()** devuelve

func(*arg1*, *arg2*)

El tipo de resultado de **operator()** está especificado por el tipo genérico **Result**. Por tanto, un objeto de esas clases puede pasarse como argumento a un adhesivo o un negador.

Ejemplo

He aquí un ejemplo que utiliza **ptr_fun()**. Crea un vector de apuntadores a carácter que señala a cadenas de caracteres. Luego utiliza la función de biblioteca estándar **strcmp()** para encontrar el apuntador que señale a "Tres". Debido a que **strcmp()** no es un objeto de función, se usa el adaptador **ptr_fun()** para permitir que el valor "Tres" se una al segundo parámetro de **strcmp()** empleando **bind2nd()**. Debido a que **strcmp()** devuelve false cuando se tiene éxito, el negador **not1()** se aplica para invertir esta condición.

Sin el uso de **ptr_fun()**, no sería posible aplicar **bind2nd()** a **strcmp()**. Es decir, debido a que **strcmp()** es una función, no es posible usarla directamente con **bind2nd()**.

```
// Usa un adaptador de apuntador a función.

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <cstring>

using namespace std;

template<class InIter>
  void mostrar_rango(const char *msj, InIter inicio, InIter final);

int main()
{
  vector<char *> v;
  vector<char *>::iterator itr;

  v.push_back("Uno");
  v.push_back("Dos");
  v.push_back("Tres");
  v.push_back("Cuatro");
  v.push_back("Cinco");

  mostrar_rango("La secuencia contiene: ", v.begin(), v.end());
  cout << endl;

  cout << "Buscando Tres en la secuencia.\n\n";

  // Usa un adaptador de apuntador a función.
  itr = find_if(v.begin(), v.end(),
        not1(bind2nd(ptr_fun(strcmp), "Tres")));

  if(itr != v.end()) {
    cout << "\u00adEncontrado!\n";
    mostrar_rango("La secuencia a la que apunta es: ", itr, v.end());
  }

  return 0;
}

// Muestra un rango de elementos..
template<class InIter>
    void mostrar_rango(const char *msj, InIter inicio, InIter final) {

  InIter itr;
  cout << msj;
  for(itr = inicio; itr != final; ++itr)
    cout << *itr << " ";
  cout << endl;
   }
```

Aquí se muestra a salida de este programa:

```
La secuencia contiene: Uno Dos Tres Cuatro Cinco

Buscando Tres en la secuencia.

¡Encontrado!
La secuencia a la que apunta es: Tres Cuatro Cinco
```

Opciones

Otro método para adaptar una función consiste en crear su propia clase de objeto de función. Haga que su **operator()** llame a la función y devuelva el resultado. Aunque es mucho menos elegante que usar un adaptador de apuntador a función, esta técnica puede ser útil en situaciones en que el resultado de la función se procesa un poco antes del uso.

El adaptador **ptr_fun()** sólo trabaja con funciones que no son miembros. La STL define adaptadores para funciones miembro, que se denominan **mem_fun()** y **mem_fun_ref()**. Se les denomina colectivamente adaptadores de función de *apuntador a miembro*.

Use los iteradores de flujo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<iterator>	istream_iterator	* ++
<iterator>	ostream_iterator	* ++
<iterator>	Istreambuf_iterator	* ++ bool equal(istreambuf_iterator<CharType, Attr> *&ob*)
<iterator>	ostreambuf_iterator	* ++ bool failed const throw()

La STL define cuatro clases que le permiten obtener iteradores a flujos de E/S. Se les suele denominar *iteradores de flujo*, y se encuentran entre algunos de los objetos de STL más interesantes, porque pormiten que un flujo de Entrada/Salida se opere de manera muy parecida a como lo hace en contenedores. Los beneficios de los iteradores de flujo son más evidentes cuando se usan con algoritmos, donde un flujo puede proporcionar entrada a alguna acción, o recibir salida de ésta. Aunque casi todas las operaciones de E/S aún utilizan operadores y funciones estándar de E/S, la

capacidad de aplicar algoritmos a flujo ofrece una nueva manera de pensar acerca de la programación de E/S. Los iteradores de flujo también pueden simplificar ciertas situaciones difíciles o tediosas de E/S. Aunque un análisis a profundidad de los iteradores de flujo es muy amplio y está más allá del alcance de este libro, en esta solución se describe el método básico necesario para usarlos.

Paso a paso

Para usar los iteradores de flujo para ingresar datos se requieren estos pasos:

1. Para crear un iterador en un flujo de entrada formado, construya un objeto de tipo **istream_iterator**, especificando el flujo de entrada.
2. Para crear un iterador a un flujo de entrada de caracteres, construya un objeto de tipo **istreambuf_iterator**, especificando el flujo de entrada.
3. Para ingresar datos del flujo, deje de hacer referencia al iterador. Luego, aumente el iterador. Esto hace que lea el siguiente elemento del flujo. Repita este proceso hasta que se lean los datos o se alcance el final del flujo.
4. El constructor predeterminado construye un iterador que indica el fin del flujo.

Para usar los iteradores de flujo para dar salida a los datos, se requieren estos pasos:

1. Para crear un iterador en un flujo de salida formado, construya un objeto de tipo **ostream_iterator**, especificando el flujo de salida.
2. Para crear un iterador a un flujo de salida de caracteres, construya un objeto de tipo **ostreambuf_iterator**, especificando el flujo de salida.
3. Para dar salida a los datos del flujo, asigne el valor mediante el iterador para dejar de hacer referencia. No es necesario aumentar el iterador. Cada asignación avanza automáticamente la salida.
4. Si ocurre un error en la salida, se devolverá la función **failed()**.

Análisis

La STL define cuatro clases de iterador de flujo. Se declaran en **<iterator>** y se muestran aquí:

Clase	Descripción
istream_iterator	Un iterador de flujo de entrada.
istreambuf_iterator	Un iterador de búfer de flujo de entrada.
ostream_iterator	Un iterador de flujo de salida.
ostreambuf_iterator	Un iterador de búfer de flujo de salida.

Una diferencia importante entre los iteradores es que **istream_iterator** y **ostream_iterator** pueden operar directamente sobre varios tipos de datos, como **int** o **double**. Los iteradores **istreambuf_iterator** y **ostreambuf_iterator** pueden operar sólo sobre caracteres. Sin embargo, la ventaja que ofrecen estos dos últimos es que le permiten realizar E/S de archivo de bajo nivel. Aquí se ofrece una revisión general de cada clase.

Los iteradores de flujo formados

Los iteradores **istream_iterator** y **ostream_iterator** pueden leer o escribir datos formados, lo que significa que pueden leer o escribir valores de carácter, de entero, de punto flotante, booleanos y de cadena. Esto los hace especialmente útiles cuando operan en flujos que contienen información legible para los seres humanos. Por ejemplo, podría usar **ostream_iterator** para escribir un entero a **cout**, o **istream_iterator** para leer una cadena de **cin**.

La clase **istream_iterator** da soporte a operaciones de iterador de entrada en un flujo. Aquí se muestra su definición de plantilla:

```
template <class T, class CharType=char, class Attr = char_traits<CharType>,
        class Diff = ptrdiff_t> class istream_iterator:
  public iterator<input_iterator_tag, T, Diff, const T *, const T &>
```

Aquí, **T** es el tipo de datos que se está transfiriendo, **CharType** es el tipo de carácter (**char** o **wchar_t**) sobre el que está operando el flujo, y **Diff** es un tipo capaz de contener la diferencia entre dos direcciones. Note que **T** es el único parámetro de tipo genérico que no es por omisión. Por tanto, debe especificar cuando se crea un **istream_operator**. Éste tiene los siguientes constructores:

```
istream_iterator()

istream_iterator(istream_type &flujo)

istream_iterator(const istream_iterator<T, CharType, Attr, Diff> &ob)
```

El primer constructor crea un iterador que indica final del flujo. Este objeto puede usarse para revisar el final de la entrada. (Es decir, se comparará igual que final del flujo.) El segundo crea un iterador al flujo especificado por *flujo*. Luego lee el primer objeto del flujo. El tipo **istream_type** es un **typedef** que especifica el tipo de flujo de entrada. La tercera forma es el constructor de copia de **istream_iterator**.

La clase **istream_iterator** define los siguientes operadores: **– >**, *, **++**. Los dos primeros actúan como se esperaría. El operador **++** requiere un poco de explicación. Cuando se usa en su forma de prefijo, **++** causa que se lea el siguiente valor del flujo de entrada. Cuando se usa en su forma de sufijo, se almacena el valor actual del valor del flujo y luego se lee el siguiente valor. En cualquier caso, para recuperar el valor, use el operador * en el iterador. Los operadores **==** y **!=** también se definen para objetos de tipo **istream_iterator**.

La clase **ostream_iterator** da soporte a las operaciones de iterador de salida en un flujo. Aquí se muestra la definición de su plantilla:

```
template <class T, class CharType=char, class Attr = char_traits<CharType> >
class ostream_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Aquí, **T** es el tipo de datos que se está transfiriendo y **CharType** es el tipo de carácter (**char** o **wchar_t**) sobre el que está operando el flujo. Observe que **T** es el único parámetro de tipo genérico que no es la opción predeterminada. Por tanto, debe especificar cuando se crea un **ostream_operator**. Éste tiene los siguientes constructores:

ostream_iterator(ostream_type &*flujo*)

ostream_iterator(ostream_type &*flujo*, const CharType **delim*)

ostream_iterator(const ostream_iterator<T, CharType, Attr> &*ob*)

El primer constructor crea un iterador al flujo especificado por *flujo*. El tipo **ostream_type** es un **typedef** que especifica el tipo de flujo de salida. La segunda forma crea un iterador al flujo especificado por *flujo* y usa los delimitadores especificados por *delim*. Los delimitadores se escriben en el flujo después de cada operación de salida. La tercera forma es el constructor de copia de **ostream_iterator**.

La clase **ostream_iterator** define los siguientes operadores: **=**, *, **++**. Para **ostream_iterator** el operador **++** no tiene efecto. Para escribir el flujo de salida, simplemente asigne un valor mediante el operador *.

Los iteradores de flujo de bajo nivel

Los iteradores de flujo de bajo nivel son **istreambuf_iterator** y **ostreambuf_iterator**. Estos iteradores leen y escriben caracteres, no datos formados. La ventaja principal de los iteradores de flujo de bajo nivel es que le dan a su programa acceso a un flujo simple de E/S byte por byte, evitando traducciones de caracteres que son posibles con los iteradores de flujo formado. Cuando se usan estos iteradores, hay una correspondencia uno a uno entre lo que está en el flujo y lo que se escribe o lee mediante el iterador.

La clase da soporte a operaciones de iterador de entrada de caracteres de bajo nivel en un flujo. Aquí se muestra su definición de plantilla:

```
template <class CharType, class Attr = char_traits<CharType> >
class istreambuf_iterator:
  public iterator<input_iterator_tag, CharType, nombretipo Attr::off_type,
                          CharType *, CharType &>
```

Aquí, **CharType** es el tipo de carácter (**char** o **wchar_t**) sobre el que está operando el flujo. **istreambuf_iterator** tiene los siguientes constructores:

istreambuf_iterator() throw()

istreambuf_iterator(istream_type &*flujo*) throw()

istreambuf_iterator(streambuf_type, **buferflujo*) throw()

El primer constructor crea un iterador que indica el fin del flujo. El segundo, uno al flujo especificado por *flujo*. El tipo **istream_type** es un **typedef** que especifica el tipo del flujo de entrada. La tercera forma crea un iterador al flujo especificado por *buferflujo*. El tipo **streambuf_type** es un **typedef** que especifica el tipo de búfer de flujo.

La clase **istreambuf_iterator** define los siguientes operadores: *, **++**. El operador **++** trabaja como se describió para **istream_iterator**. Para leer un carácter de la cadena, aplique * al iterador.

Para pasar al siguiente carácter, aumente el iterador. Los operadores **==** y **!=** también se definen para objetos de tipo **istreambuf_iterator**.

istreambuf_iterator define la función miembro **equal()**, que se muestra aquí:

bool equal(istreambuf_iterator<CharType, Attr> &*ob*)

Su operación va un poco en contra de la intuición. Devuelve true si el iterador que invoca y *ob* señalan al final del flujo. También devuelve true si ambos operadores no señalan al final del flujo. No es necesario que señalen a lo mismo. Devuelve false, de otra manera. Los operadores **==** y **!=** funcionan igual.

La clase **ostreambuf_iterator** da soporte a operaciones de iterador de salida de caracteres de bajo nivel en un flujo. Aquí se muestra su definición de plantilla:

template <class CharType, class Attr = char_traits<CharType> >
class ostreambuf_iterator:
  public iterator<output_iterator_tag, void, void, void, void>

Aquí, **CharType** es el tipo de carácter (**char** o **wchar_t**) sobre el que está operando el flujo. **ostreambuf_iterator** tiene los siguientes constructores:

ostreambuf_iterator(ostream_type &*flujo*) throw()

ostreambuf_iterator(streambuf_type, **buferflujo*) throw()

La primera crea un iterador al flujo especificado por *flujo*. El tipo **ostream_type** es un **typedef** que especifica el tipo de flujo de entrada. La segunda forma crea un iterador que usa el búfer de flujo especificado por *buferflujo*. El tipo **streambuf_type** es un **typedef** que especifica el tipo de búfer de flujo.

La clase **ostreambuf_iterator** define los siguientes operadores: **=**, *****, **++**. El operador **++** no tiene efecto. Para escribir un carácter en el flujo, simplemente asigne un valor mediante el operador *.

La clase **ostreambuf_iterator** también define la función **failed()**, como se muestra aquí:

bool failed() const throw()

Devuelve false si no ha ocurrido una falla, y true de otra manera.

Ejemplo

En el siguiente programa se demuestra la manera en que **istream_iterator** y **ostream_iterator** pueden usarse para leer de **cin** y escribir en **cout**. Aunque por lo general usará los iteradores de flujo para este fin, el programa ilustra claramente la manera en que funcionan. Por supuesto, el poder real de los iteradores de flujo se encuentra cuando se usan con algoritmos, lo que se demuestra con el ejemplo adicional que sigue.

```
// Usa istream_iterator y ostream_ierator para leer de cin y escribir en cout.

#include <iostream>
#include <iterator>
#include <string>
#include <vector>
```

```
using namespace std;

int main()
{
  unsigned i;
  double d;
  string cad;
  vector<int> vi;
  vector<double> vd;
  vector<string> vs;

  // Usa istream_iterator para leer de cin.

  // Crea un iterador de flujo de entrada para enteros.
  cout << "Ingrese algunos enteros, ingrese 0 para detener.\n";
  istream_iterator<int> itr_ent(cin);
  do {
    i = *itr_ent; // lee el siguiente entero
    if(i != 0) {
      vi.push_back(i); // lo almacena
      ++itr_ent; // ingresa el siguiente entero
    }
  } while (i != 0);

  // Crea un iterador de flujo de entrada para doubles
  cout << "Ingrese algunos doubles, ingrese 0 para detener.\n";
  istream_iterator<double> itr_double(cin);
  do {
    d = *itr_double; // lee el siguiente double
    if(d != 0.0) {
      vd.push_back(d); // lo almacena
      ++itr_double; // ingresa el siguiente double
    }
  } while (d != 0.0);

  // Crea un iterador de flujo de entrada para cadena.
  cout << "Ingrese algunas cadenas, ingrese 'salir' para detener.\n";
  istream_iterator<string> itr_cadena(cin);
  do {
    cad = *itr_cadena; // lee la siguiente cadena
    if(cad != "salir") {
      vs.push_back(cad); // la almacena
      ++itr_cadena;
    }
  } while (cad != "salir"); // ingresa la siguiente cadena

  cout << endl;

  cout << "Esto es lo que ingres\u00a2:\n";
  for(i=0; i < vi.size(); i++) cout << vi[i] << " ";
  cout << endl;

  for(i=0; i < vd.size(); i++) cout << vd[i] << " ";
  cout << endl;
```

```
  for(i=0; i < vs.size(); i++) cout << vs[i] << " ";


  // Ahora, usa ostream_iterator para escribir a cout.

  // Crea un iterador de salida para cadenas.
  ostream_iterator<string> salida_itr_cadena(cout);
  *salida_itr_cadena = "\n";
  *salida_itr_cadena = string("\nSe trata de una cadena\n");
  *salida_itr_cadena = "Aqu\u00a1 hay otra.\n";

  // Crea un iterador de salida para int.
  ostream_iterator<int> salida_itr_ent(cout);
  *salida_itr_ent = 10;
  *salida_itr_cadena = " ";
  *salida_itr_ent = 15;
  *salida_itr_cadena = " ";
  *salida_itr_ent = 20;

  *salida_itr_cadena = "\n";

  // Crea un interador de salida para bool.
  ostream_iterator<bool> salida_itr_bool(cout);
  *salida_itr_bool = true;
  *salida_itr_cadena = " ";
  *salida_itr_bool = false;

  return 0;
}
```

Aquí se muestra una ejecución de ejemplo:

```
Ingrese algunos enteros, ingrese 0 para detener.
1 2 3 0
Ingrese algunos doubles, ingrese 0 para detener.
1.1 2.2 3.3 0.0
Ingrese algunas cadenas, ingrese 'salir' para detener.
Se trata de una prueba
salir

Esto es lo que ingresó:
1 2 3
1.1 2.2 3.3
Se trata de una prueba

Se trata de una cadena
Aquí hay otra.
10 15 20
1 0
```

Ejemplo adicional: cree un filtro de archivo de STL

Aunque el uso de iteradores de flujo para escribir en la consola o leer de ésta, como se hace en el ejemplo anterior, es un uso intrigante, no muestra su real capacidad. No es hasta que se combinan los iteradores de flujo con algoritmos cuando emerge su verdadero potencial. En el siguiente programa se muestra un ejemplo de la manera en que se mejora un proyecto de programación que, de otro modo, sería tedioso.

Como se explicó, los iteradores de flujo de bajo nivel operan sobre caracteres, pasando por alto la inclusión en búfer y las posibles traducciones de caracteres que podrían ocurrir con los iteradores de flujo de alto nivel. Esto los hace perfectos para manipular el contenido de un archivo mediante un algoritmo. La operación sobre el contenido de un archivo mediante uno o más algoritmos de STL es un concepto poderoso. A menudo resulta posible implementar una operación de archivo sofisticada que normalmente requeriría varias líneas de código en una simple llamada a un algoritmo. En el ejemplo expuesto aquí se demuestra esto. Implementa un filtro de archivo relativamente simple.

Un *filtro de archivo* es un programa de utilería que elimina o reemplaza información específica cuando se copia un archivo. El siguiente programa es un ejemplo simple de este tipo de filtro. Copia un archivo y en el proceso reemplaza un carácter con otro. El nombre del archivo, el carácter que se reemplazará y el carácter de reemplazo se especifican en la línea de comandos. Para manejar el reemplazo, se usan los iteradores de flujo de carácter y el algoritmo **replace_copy()**.

```
// Usa istreambuf_iterator, ostreambuf_iterator y replace_copy()
// para filtrar un archivo.

#include <iostream>
#include <fstream>
#include <iterator>
#include <algorithm>

using namespace std;

int main(int argc, char *argv[])
{
  if(argc != 5) {
    cout << "Uso: Reemplazar entrada salida antiguocar nuevocar\n";
    return 1;
  }

  ifstream entrada(argv[1]);
  ofstream salida(argv[2]);

  // Se asegura de que los archivos se abren con éxito.
  if(!entrada.is_open()) {
    cout << "No se puede abrir el archivo de entrada.\n";
    return 1;
  }
  if(!salida.is_open()) {
    cout << "No se puede abrir el archivo de salida.\n";
    return 1;
  }

  // Crea iteradores de flujo.
```

```
  istreambuf_iterator<char> itr_entrada(entrada);
  ostreambuf_iterator<char> itr_salida(salida);

  // Copia el archivo, reemplazando caracteres en el proceso.
  replace_copy(itr_entrada, istreambuf_iterator<char>(),
               itr_salida, *argv[3], *argv[4]);

  // Los destructores de ofstream e ifstream llaman a close(),
  // así que las siguientes llamadas no son necesarias en este caso.
  // Sin embargo, para evitar confusión, en este libro se cierran
  // explícitamente todos los archivos.
  entrada.close();
  salida.close();

  return 0;
}
```

Para comprender los efectos del programa, suponga un archivo llamado **Prueba.dat** que contiene lo siguiente:

```
Ésta es una prueba que utiliza el iterador de flujo con un algoritmo.
```

A continuación, suponiendo que el programa se llama Reemplazar, después de que se ejecuta esta línea de comandos:

```
C:>Reemplazar Prueba.dat Prueba2.dat t X
```

Todos los casos de **t** se reemplazarán con **X** cuando se copie **Prueba.dat** en **Prueba2.dat**. Por tanto, el contenido de éste será:

```
EsXa es una prueba que uXiliza el iXerador de flujo con un algoriXmo.
```

Observe que una vez que los archivos están abiertos, sólo se requiere una instrucción, la llamada a **replace_copy()**, para copiar el archivo, reemplazando en el proceso todos los casos de un carácter con otro. Para hacer esto sin el uso de **replace_copy()** se requerirían varias líneas de código. Si lo piensa, queda claro que los algoritmos de STL ofrecen una solución elegante a muchos tipos de tareas de manejo de archivos. Ésta es una de las capacidades más importantes, pero subutilizadas de la STL.

Opciones

Los iteradores de flujo son realmente una característica única. No son una opción directamente paralela. Si quiere operar en flujos mediante iteradores, lo hará mediante los iteradores de flujo que se acaban de describir. Por supuesto, siempre podría crear sus propias implementaciones personalizadas, pero apenas sería (si acaso) una razón para ello. Los iteradores de flujo ofrecen una opción poderosa al método "normal" para E/S, como los operadores y los manipuladores de E/S.

Para el caso de soluciones que se concentran en el sistema de E/S de C++, consulte el capítulo 5.

Use los adaptadores de iterador de inserción

Componentes clave		
Encabezados	**Clases**	**Funciones**
<iterator>		template <class Cont> front_insert_iterator<Cont> front_inserter(Cont &*cnt*) template <class Cont> back_insert_iterator<Cont> back_inserter(Cont &*cnt*) template <class Continuación, class OutIter> insert_iterator<Cont> inserter(Cont &*cnt*, OutIter *itr*)

La STL define tres adaptadores de iterador que se usan para obtener un iterador que inserta, en lugar de sobreescribir, elementos en un contenedor. A estos adaptadores se les denomina **back_inserter()**, **front_inserter()** e **inserter()**. Se declaran en **<iterator>**. En esta solución se muestra cómo usarlos.

Los adaptadores de iterador de inserción son herramientas muy útiles. Para comprender por qué, considere los dos comportamientos asociados con iteradores. En primer lugar, cuando se usan iteradores normales para copiar un elemento en un contenedor, el contenido actual del rango de destino se sobreescribe. Es decir, el elemento que se está copiando no se inserta en el contenedor, sino que reemplaza (es decir, sobreescribe) al elemento anterior. Por tanto, no se preserva el contenido anterior del contenedor de destino. En segundo lugar, cuando se copian elementos en un contenedor mediante un iterador normal, es posible sobreescribir el final del contenedor. Recuerde que un contenedor no aumentará automáticamente su tamaño cuando se usa como el destino de un algoritmo; debe tener el tamaño suficiente para acomodar el número de elementos que recibirá antes de que una operación de copia tenga lugar. Un iterador de inserción le permite modificar estos dos comportamientos.

Cuando se agrega un elemento a un contenedor mediante un iterador de inserción, el elemento se inserta en la ubicación a la que señala el iterador, y cualquier elemento restante se mueve para hacer espacio al nuevo elemento. Por tanto, se preserva el contenido original del contenedor. Si es necesario, se aumenta el tamaño del contenedor para acomodar el elemento insertado. No es posible sobreescribir el final del contenedor de destino.

Paso a paso

Para adaptar un iterador para operaciones de inserción se requieren estos pasos:

1. Para obtener un iterador que puede insertar en cualquier punto de un contenedor, llame a **inserter()**, especificando el contenedor y un iterador al punto en que quiere que ocurra la inserción.
2. Para obtener un iterador que puede insertar al final de un contenedor, llame a **back_inserter()**, especificando el contenedor.
3. Para obtener un iterador que puede insertar al frente de un contenedor, llame a **front_inserter()**, especificando el contenedor.

Análisis

Para obtener un iterador que puede insertar elementos en cualquier punto de un contenedor, use la función **inserter()**, que se muestra aquí:

template <class Continuación, class OutIter> insert_iterator<Cont>
inserter(Cont &*cnt*, OutIter *itr*)

Aquí, *cnt* es el contenedor sobre el que se está operando e *itr* señala a la ubicación en que ocurrirán las inserciones. Devuelve un iterador de tipo **insert_iterator**. La clase **insert_iterator** encapsula un iterador de salida que inserta objetos en un contenedor.

Para obtener un iterador que pueda insertar elementos al final de un contenedor, llame a **back_inserter()**. Aquí se muestra:

template <class Cont> back_insert_iterator<Cont> back_inserter(Cont &*cnt*)

El contenedor que recibe las inserciones se pasa vía *cnt*. Devuelve un iterador de tipo **back_insert_iterator**. La clase **back_insert_iterator** encapsula un iterador de salida que inserta objetos al final de un contenedor. El contenedor que recibe debe dar soporte a la función **push_back()**.

Para obtener un iterador que puede insertar elementos al frente de un contenedor, llame a **front_inserter()**. Aquí se muestra:

template <class Cont> front_insert_iterator<Cont> front_inserter(Cont &*cnt*)

El contenedor que recibe las inserciones se pasa vía *cnt*. Devuelve un iterador de tipo **front_insert_iterator**. La clase **front_insert_iterator** encapsula un iterador de salida que inserta objetos al frente del contenedor. El contenedor que recibe debe dar soporte a la función **push_front()**. Esto significa que un **vector**, por ejemplo, no puede ser el destino de **front_insert_iterator**.

Ejemplo

Cada uno de estos iteradores insertan en el contenido de un contenedor, en lugar de sobreescribirlo. En el siguiente ejemplo se demuestra cada tipo de iterador de inserción al copiar el contenido de una **deque** en otra. Debido a que se usan los iteradores de inserción, la **deque** original no se sobreescribe. En cambio, los nuevos elementos se insertan en él.

```
// Usa adaptadores de iterador de inserción para insertar
// una deque en otra mediante el algoritmo copy().

#include <iostream>
#include <iterator>
#include <deque>
#include <string>

using namespace std;

void mostrar(const char *msj, deque<string> dq);

int main()
{
```

```
  deque<string> dq, dq2, dq3, dq4;

  dq.push_back("Los");
  dq.push_back("iteradores");
  dq.push_back("son");
  dq.push_back("une");
  dq.push_back("a");
  dq.push_back("STL.");

  dq2.push_back("el");
  dq2.push_back("pegamento");
  dq2.push_back("que");

  dq3.push_back("Hasta");
  dq3.push_back("el");
  dq3.push_back("final.");

  dq4.push_back("frente.");
  dq4.push_back("el");
  dq4.push_back("En");

  cout << "Tama\u00a4o original de dq: " << dq.size() << endl;
  mostrar("Contenido original de dq:\n", dq);
  cout << endl;

  // Usa un insert_iterator para insertar dq2 en dq.
  copy(dq2.begin(), dq2.end(), inserter(dq, dq.begin()+3));

  cout << "Tama\u00a4o de dq tras insertar dq2: ";
  cout << dq.size() << endl;
  mostrar("Contenido de dq tras insertar dq2:\n", dq);
  cout << endl;

  // Usa un back_insert_iterator para insertar dq3 en dq.
  copy(dq3.begin(), dq3.end(), back_inserter(dq));

  cout << "Tama\u00a4o de dq tras insertar dq3: ";
  cout << dq.size() << endl;
  mostrar("Contenido de dq tras insertar dq3:\n", dq);
  cout << endl;

  // Usa un front_insert_iterator para insertar dq4 en dq.
  copy(dq4.begin(), dq4.end(), front_inserter(dq));

  cout << "Tama\u00a4o de dq tras insertar dq4: ";
  cout << dq.size() << endl;
  mostrar("Contenido de dq tras insertar dq4:\n", dq);

  return 0;
}

// Despliega el contenido de una deque<string>.
void mostrar(const char *msj, deque<string> dq) {
```

```
  cout << msj;
  for(unsigned i=0; i < dq.size(); ++i)
    cout << dq[i] << " ";
  cout << "\n";
}
```

He aquí la salida del programa.

```
Tamaño original de dq: 6
Contenido original de dq:
Los iteradores son los que se unen a STL.

Tamaño de dq tras insertar dq2: 9
Contenido de dq tras insertar dq2:
Los iteradores son el pegamento que une a STL.

Tamaño de dq tras insertar dq3: 12
Contenido de dq tras insertar dq3:
Los iteradores son el pegamento que une a STL. Hasta el final.

Tamaño de dq tras insertar dq4: 15
Contenido de dq tras insertar dq4:
En el frente. Los iteradores son el pegamento que une a STL. Hasta el final.
```

Como puede ver, **dq2** se insertó en medio, **dq3** se insertó al final y **dq4** se insertó al frente de **dq**. En el proceso, se aumentó automáticamente el tamaño de **dq** para contener los elementos adicionales. Si no se ha usado un iterador de inserción, se habría sobrescrito el contenido original de **dq**.

Opciones

Los adaptadores de iterador de inserción suelen usarse cuando un algoritmo copia el resultado de una operación en otro contenedor. Esta situación ocurre con algoritmos como **replace_copy()**, **reverse_copy()**, **remove_copy()**, etc. También ocurre con casi todo el conjunto de algoritmos. Al usar un adaptador de iterador de inserción, puede habilitar estos algoritmos para insertar el resultado en el contenedor de destino, en lugar de sobreescribir los elementos existentes. Esta capacidad expande en gran medida los tipos de problemas a los que pueden aplicarse estos algoritmos.

CAPÍTULO 5

Trabajo con E/S

En este capítulo se presentan soluciones que utilizan el sistema de E/S de C++. Como todos los lectores saben, E/S es una parte integral de casi todos los proyectos de programación. Como resultado, casi todos los lenguajes de computación tienen importantes subsistemas dedicados a él, y C++ no es una excepción. La biblioteca de E/S de C++ tiene una enorme cantidad de opciones, pero resulta flexible y fácil de usar. También es extensible. Con base en una jerarquía compleja de clases, el sistema de E/S ofrece al programador un marco conceptual bien organizado que puede aplicarse a casi cualquier situación.

Debido a la importancia de E/S, es un tema que genera muchas preguntas de tipo "¿Cómo hacer?", tanto de novatos como de profesionales experimentados. Por supuesto, dado el tamaño y el alcance de la biblioteca de E/S, no es posible presentar soluciones que cubran todos los aspectos y detalles de este poderoso subsistema. Para ello, se requeriría un libro completo. En cambio, en este capítulo se responden varias de las preguntas más comunes. Como era de esperarse, su principal eje está en el manejo de archivos, incluidas soluciones que muestran cómo leer y escribir datos, realizar acceso aleatorio y detectar errores. En otras soluciones se describe cómo crear manipuladores personalizados de E/S, sobrecargar los operadores de E/S y usar un flujo de cadena.

Como elemento adicional, se incluye una solución que describe el núcleo del sistema de E/S heredado del lenguaje C. Debido a que C++ estaba integrado en C, C++ también incluye todo el sistema de archivo de C. Aunque no se recomienda para programas de C++, el sistema de archivos de C aún está muy difundido en código C heredado. La solución basada en C será interesante para cualquier persona que necesite mantener código C o llevarlo a C++.

Otro tema importante es que, a pesar de que el sistema de E/S también maneja la formación de datos para entrada y operaciones, este tema se explora de manera independiente en el capítulo 6. El eje de este capítulo está en la base del E/S de C++.

He aquí las soluciones contenidas en este capítulo:

- Escriba datos formados en un archivo de texto
- Lea datos formados de un archivo de texto
- Escriba datos binarios sin formar en un archivo
- Lea datos binarios sin formar de un archivo
- Use **get()** y **getline()** para leer un archivo
- Lea un archivo y escriba en él
- Detección de EOF
- Use excepciones para detectar y manejar errores de E/S

- Use E/S de archivo de acceso aleatorio
- Revise un archivo
- Use los flujos de cadena
- Cree insertadores y extractores personalizados
- Cree un manipulador sin parámetros
- Cree un manipulador con parámetros
- Obtenga o establezca una configuración regional y de idioma de flujo
- Use el sistema de archivos de C
- Cambie el nombre de un archivo y elimínelo

NOTA *Como se explicó en* Use los iteradores de flujo *en el capítulo 4, es posible usar algoritmos de STL junto con iteradores de flujo para realizar una amplia variedad de tareas de E/S y manejo de archivos. En algunos casos, el uso de iteradores de flujo y de algoritmos simplifica algunas tareas que, de otra manera, serían complicadas. Sin embargo, el eje de este capítulo está en el sistema de E/S de C++. Por ello, en las soluciones no se usan los algoritmos de STL. Sólo recuerde que los iteradores de flujo y los algoritmos de STL ofrecen una opción interesante que podría ser útil en algunos casos.*

Revisión general de E/S

El sistema de E/S de C++ está basado en una colección coherente, interrelacionada de clases que proporcionan la funcionalidad necesaria para realizar operaciones de entrada y salida eficientes en diversos dispositivos, incluidos la consola y los archivos de disco. Aunque ninguna parte del sistema de E/S resulta difícil de dominar, es muy grande y depende de varias clases y muchas funciones. Por tanto, aquí se presenta una breve revisión del sistema de E/S de C++. Este análisis es suficiente para los objetivos de las soluciones de este capítulo, pero los lectores que deseen realizar programación avanzada de E/S, como derivar clases para manejar dispositivos especializados, necesitarán estudiar el sistema de E/S con un detalle mayor.

Flujos de C++

La base del sistema de E/S de C++ es el *flujo*. Un flujo es una abstracción que produce o consume información. Todos los flujos se comportan de la misma manera, aunque los dispositivos físicos reales a los que se vinculan sean diferentes. Esto significa que la manera en que opera un tipo de flujo es la misma para todos los flujos. Por ejemplo, la función **pull()** puede usarse para escribir en la pantalla, en un archivo de disco o en la impresora.

En su forma más común, un flujo es una interfaz lógica con un archivo. De acuerdo con la definición del término *archivo* en C++, puede aludir a un archivo de disco, la pantalla, el teclado, un puerto, un archivo en cinta, etc. Aunque los archivos tienen diferentes formas y capacidades, todos los flujos son iguales. La ventaja de este método es que para el programador, un dispositivo de hardware será muy parecido a otro. El flujo proporciona una interfaz consistente.

Un flujo está vinculado a un archivo mediante una operación *abierta*. Un flujo se disocia de un archivo mediante una operación de *cierre*.

Hay dos tipos de flujos: *de texto* y *binario*. Un flujo de texto se usa con información legible para el ser humano. En un flujo de texto, es posible que se realice alguna traducción de caracteres. Por ejemplo, cuando se da salida al carácter de nueva línea, puede convertirse en una secuencia retorno de carro/avance de línea. Por esto, tal vez no haya correspondencia uno a uno entre lo que se

envía al flujo y lo que se escribe en el archivo. Un flujo binario puede usarse con cualquier tipo de datos. No ocurrirá traducción de caracteres, y hay correspondencia uno a uno entre lo que se envía al flujo y lo que en realidad contiene el archivo.

Un concepto adicional que se debe comprender es el de *ubicación actual*. Ésta (a la que también se denomina *posición actual*) es la ubicación, en un flujo, donde ocurrirá la operación de E/S. Por ejemplo, considere una situación en que un flujo está vinculado a un archivo. Si éste tiene 100 bytes de largo y se ha leído la mitad del archivo, la siguiente operación de lectura ocurrirá en el byte 50, que es la ubicación actual.

Para resumir: en C++, la E/S se realiza mediante una interfaz lógica llamada flujo. Todos los flujos tienen propiedades similares, y cada flujo se opera con las mismas funciones de E/S, sin importar qué tipo de archivo está relacionado con él. Un archivo es la entidad física real que contiene los datos. Aunque los archivos sean diferentes, los flujos no. (Por supuesto, es posible que algunos dispositivos no den soporte a todas las operaciones, como las de acceso aleatorio, de modo que sus flujos asociados no darán tampoco soporte a esas operaciones.)

Las clases de flujo de C++

El sistema de E/S de C++ está construido a partir de un sistema más bien complejo de clases de plantillas. Aquí se muestran estas clases.

Clase	Propósito
basic_ios	Proporciona operaciones de E/S de propósito general.
basic_streambuf	Soporte de nivel inferior para E/S.
basic_istream	Soporte para operaciones de entrada. Hereda **basic_ios**.
basic_ostream	Soporte para operaciones de salida. Hereda **basic_ios**.
basic_iostream	Soporte para operaciones de entrada/salida. Hereda **basic_istream** y **basic_ostream**.
basic_filebuf	Soporte de bajo nivel para E/S de archivo. Hereda **basic_streambuf**.
basic_ifstream	Soporte para entrada de archivo. Hereda **basic_istream**.
basic_ofstream	Soporte para salida. Hereda **basic_ostream**.
basic_fstream	Soporte para entrada/salida de archivos. Hereda **basic_iostream**.
basic_stringbuf	Soporte de bajo nivel para E/S de cadena. Hereda **basic_streambuf**.
basic_istringstream	Soporte para entrada cadena. Hereda **basic_istream**.
basic_ostringstream	Soporta para salida de cadena. Hereda **basic_ostream**.
basic_stringstream	Soporte para entrada/salida de cadena. Hereda **basic_iostream**.

La clase **ios_base**, que no es de plantilla, también forma parte de la jerarquía de clases de E/S. Proporciona definiciones para varios elementos del sistema de E/S que no dependen de parámetro de plantilla.

El sistema de E/S de C++ utiliza dos jerarquías de clase de plantilla relacionadas pero diferentes. La primera se deriva de la clase de E/S de bajo nivel llamada **basic_streambuf**, que requiere el encabezado **<streambuf>**. Esta clase proporciona las operaciones básicas de entrada y salida de bajo nivel de un búfer de flujo, que proporciona el soporte básico para todo el sistema de E/S de C++. Cada flujo contiene un objeto de **basic_streambuf**, aunque por lo general no necesitará tener acceso directo a él. Las clases **basic_filebuf** y **basic_stringbuf** derivan de **basic_streambuf**. A menos que esté haciendo programación avanzada de E/S, no necesitará usar directamente **basic_streambuf** ni su subclase. En cambio, utilizará sus características mediante funciones definidas por las clases de flujo.

La jerarquía de clase con la que estará trabajando de manera más común se deriva de **basic_ios**. Está declarada en el encabezado **<ios>**. Se trata de una clase de E/S de alto nivel que define características comunes para todos los flujos, como revisión de errores e información de estado. Una clase base para **basic_ios** es **ios_base**. Como se explicó, define varias funciones sin plantilla usadas por **basic_ios**, como formación. La clase **basic_ios** se usa como base para varias clases derivadas, incluidas **basic_istream**, **basic_ostream** y **basic_iostream**. Estas clases proporcionan la funcionalidad esencial necesaria para flujos con capacidad de entrada, salida y entrada/salida, respectivamente.

Las clases de E/S reciben parámetros para los tipos de caracteres sobre los que actúan y para los rastros asociados con esos caracteres. Por ejemplo, he aquí la especificación de plantilla para **basic_ios**:

```
template <class CharType, class CharTraits = char_traits<CharType> >
        class basic_ios: public ios_base
```

Aquí **CharType** especifica el tipo de carácter (como **char** o **wchar_t**) y **CharTraits** especifica un tipo que describe el atributo de **CharType**. Observe que la opción predeterminada de **CharTraits** es **char_traits<CharType>**. El tipo genérico **char_traits** es una clase de utilería que define los atributos asociados con un carácter.

Para realizar E/S de archivo, debe incluir el encabezado **<fstream>** en su programa. Define varias clases, incluidas **basic_ifstream**, **basic_ofstream** y **basic_fstream**. Estas clases son derivadas de **basic_istream**, **basic_ostream** y **basic_iostream**, respectivamente. Recuerde que estas tres últimas clases derivan de **basic_ios**, de modo que los flujos de archivos también tienen acceso a todas las operaciones definidas por **basic_ios**.

El sistema de E/S también da soporte al uso de una **string** como origen o destino de operaciones de E/S. Para ello, usará las clases de flujo de cadena. El soporte de bajo nivel es proporcionado por **basic_stringbuf**, que deriva de **basic_streambuf**. Las clases de flujo de cadena son **basic_istringstream**, **basic_ostringstream** y **basic_stringstream**. Estas clases derivan de **basic_istream**, **basic_ostream** y **basic_iostream**, respectivamente. Crean flujos de cadena con opciones de entrada, salida y entrada/salida.

Como se mencionó, cada flujo tiene asociado un objeto derivado de **basic_streambuf**, pero casi nunca necesitará interactuar directamente con el objeto de **basic_strambuf**. En cambio, en casi todos los casos (incluidas todas las soluciones de este capítulo), utilizará las características proporcionadas por las clases de flujo, que se derivan de **basic_ios**. En las siguientes secuencias se ofrece una breve revisión general de cada una. En las soluciones individuales se describen a profundidad las características que utilizan. Empezaremos con **ios_base**.

ios_base

La clase **ios_base** encapsula los aspectos de E/S que son comunes a todos los flujos y que no dependen de parámetros de plantilla. Requiere el encabezado **<ios>**. La clase **ios_base** define varios tipos y funciones. He aquí los tipos usados en este libro:

fmtflags	Máscara de bits que determina el formato de la información a la que se da salida.
iostate	Máscara de bits que indica el estado de un flujo.
openmode	La máscara de bits que indica cómo se abre un archivo.
seekdir	Una enumeración que controla la manera en que se maneja la E/S de acceso aleatorio.

He aquí una muestra de sus métodos:

flags()	Obtiene o establece todas las marcas de formato.
setf()	Obtiene o establece marcas específicas de formato.
unsetf()	Limpia una o más marcas de formato.
precision()	Obtiene o establece la precisión.
width()	Obtiene o establece el ancho del campo.
imbue()	Establece la configuración regional y de idioma.
getloc()	Obtiene la configuración regional y de idioma.

basic_ios

La clase **basic_ios** hereda **ios_base** y luego define las características relacionadas con plantillas que son comunes a todos los flujos. Utiliza el encabezado **<ios>**. Define los siguientes **typedefs** que indican tipo (y, por tanto, el tamaño) de varios tipos usados por el sistema de E/S. Aquí se muestran:

char_type	El tipo de carácter.
int_type	El tipo de entero.
pos_type	Un tipo que puede representar una posición dentro de un archivo.
off_type	Un tipo que puede representar un desplazamiento dentro de un archivo.
traits_type	Un tipo que describe los atributos de un carácter.

La clase **basic_ios** también define varias funciones. Aquí se muestran las usadas en este capítulo:

clear()	Limpia las marcas de error de E/S.
exceptions()	Establece u obtiene los errores que pueden causar el lanzamiento de una excepción.
eof()	Devuelve true si se alcanza el final del archivo.
bad()	Devuelve true si ha ocurrido un error no recuperable.
fail()	Devuelve true si ha ocurrido un error.
fill()	Obtiene o establece el carácter de relleno usado para llenar un flujo.
good()	Devuelve true si no ha ocurrido un error.
rdstate()	Obtiene una máscara de bits que contiene las marcas de estado de E/S.
setstate()	Establece una o más marcas de E/S.

Observe que muchos de éstos se relacionan con las marcas que representan el estado de un flujo de E/S. Se usan para detectar y manejar errores en condición de final de archivo. (Las técnicas de manejo de errores se describen más adelante, en esta misma revisión general.)

La clase **basic_ios** también define los operadores * y **!** que pueden aplicarse a un flujo. El operador * devuelve un apuntador nulo si un flujo es erróneo y, de lo contrario, uno no nulo. El **!** devuelve el resultado de **fail()**. Por tanto, si no han ocurrido errores, **!** devuelve false. De otra manera, devuelve true.

basic_istream

La clase **basic_istream** hereda **basic_ios** y define la funcionalidad común a todos los flujos de entrada. Por tanto, **basic_istream** es el eje de todos los flujos de entrada. Requiere el encabezado **<istream>**.

La clase **basic_istream** define el extractor **>>**, que lee datos formados del flujo de entrada. Este operador está sobrecargado para todos los tipos integrados. Varias funciones están definidas por **basic_istream**. Aquí se muestran las usadas en este capítulo:

gcount()	Devuelve el número de caracteres leído por la última operación de entrada.
get()	Lee y elimina uno o más caracteres del flujo de entrada.
getline()	Lee y elimina una línea de texto del flujo de entrada.
ignore()	Lee y descarta caracteres del flujo de entrada.
peek()	Lee, pero no elimina, un carácter del flujo de entrada.
putback()	Devuelve un carácter al flujo de entrada.
read()	Lee y elimina caracteres del flujo de entrada.
seekg()	Establece la posición del archivo para entrada.
tellg()	Devuelve la posición actual en el flujo de entrada.
unget()	Devuelve al flujo de entrada el último carácter leído del flujo.

basic_ostream

La clase **basic_ostream** hereda **basic_ios** y define la funcionalidad común a todos los flujos de salida. Por tanto, **basic_ostream** es una clase de base para **basic_ofstream**, por ejemplo. Requiere el encabezado **<ostream>**.

La clase **basic_ostream** define el insertador **<<**, que escribe datos formados en el flujo de salida. Este operador está sobrecargado para todos los tipos de entrada. Varias funciones están definidas por **basic_ostream**. Aquí se muestran las usadas en este capítulo:

flush()	Escribe datos incluidos en el búfer al flujo de salida.
put()	Escribe un carácter en el flujo de salida.
seekp()	Establece la posición actual del archivo para salida.
tellp()	Devuelve la posición actual del flujo de salida.
write()	Escribe caracteres al flujo de salida.

basic_iostream

La clase **basic_iostream** hereda **basic_istream** y **basic_ostream**. Por tanto, encapsula las características de un flujo que tienen opciones de entrada y salida.

basic_ifstream

La clase **basic_ifstream** hereda **basic_istream** y agrega la funcionalidad necesaria para la entrada de archivo. Requiere el encabezado **<fstream>**. Define cuatro funciones; de ellas, en este capítulo se usan las siguientes tres:

close()	Cierra un archivo, liberando cualquier recurso del sistema usado por ese archivo.
is_open()	Devuelve true si un archivo está abierto.
open()	Abre un archivo para entrada. También es posible usar un constructor **basic_ifstream** para abrir un archivo.

basic_ofstream

La clase **basic_ofstream** hereda **basic_ostream** y agrega la funcionalidad requerida para salida de archivos. Necesita el encabezado **<fstream>**. Define cuatro funciones; de ellas, en este capítulo se utilizan las tres siguientes:

close()	Cierra un archivo, liberando cualquier recurso del sistema usado por ese archivo.
is_open()	Devuelve true si un archivo está abierto.
open()	Abre un archivo para entrada. También es posible usar un constructor **basic_ofstream** para abrir un archivo.

basic_fstream

La clase **basic_fstream** hereda **basic_iostream**. Por tanto, contiene la funcionalidad requerida para entrada y salida de archivos. Necesita el encabezado **<fstream>**. Define cuatro funciones; de ellas, en este capítulo se utilizan las tres siguientes:

close()	Cierra un archivo, liberando cualquier recurso del sistema usado por ese archivo.
is_open()	Devuelve true si un archivo está abierto.
open()	Abre un archivo para entrada. También es posible usar un constructor **basic_fstream** para abrir un archivo.

Las especializaciones de clases relacionadas con los flujos

Como ya se explicó, las clases relacionadas con flujo en C++ son plantillas que toman el tipo de carácter y sus atributos como parámetros de tipo. Esto significa que el sistema de E/S puede operar sobre flujos basados en caracteres de ocho bits y en caracteres extendidos. Como conveniencia, la biblioteca de E/S crea dos especializaciones de jerarquías de clases de plantilla: una para **char** y otra para **wchar_t**. Al usar estas especializaciones, no tendrá que proporcionar de manera continua los parámetros de tipo cuando se declaran y usan objetos de flujo.

He aquí una lista de los nombres de clases de plantilla a sus versiones **char** y **wchar_t.**

Clase de plantilla	Especialización para char	Especialización para wchar_t
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream
basic_fstream	fstream	wfstream
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream
basic_istringstream	istringstream	wistringstream
basic_ostringstream	ostringstream	wostringstream
basic_stringstream	stringstream	wstringstream
basic_streambuf	streambuf	wstreambuf
basic_filebuf	filebuf	wfilebuf
basic_stringbuf	stringbuf	wstringbuf

Observe que los nombres empleados para los flujos de **char** son simplemente el nombre de la clase de plantilla sin la parte **basic_**. Por ejemplo, la versión para **char** de **basic_ifstream** es **ifstream**. La versión de **basic_ios** es **ios**. Los flujos de carácter extendido usan el mismo método, pero con la **w** agregada.

Las especializaciones son los nombres que suelen usarse cuando se programa, porque crean automáticamente el tipo de flujo deseado, en lugar de tener que especificar un argumento de tipo. Por ejemplo, por lo general usará **ifstream** para abrir un archivo, no **basic_ifstream<char>**, y normalmente especificará **ios**, no **basic_ios<char>**. Así, no sólo está utilizando la especialización, también asegura que se creen en todos los casos los objetos de flujo apropiados, con lo que se evitan errores.

De los dos tipos de flujo, los de **char** se usan con más frecuencia. Una razón para esto es que en C++, un **char** corresponde a un byte, y en el nivel más bajo, toda la E/S está basada en bytes. Por tanto, a menos que explícitamente esté operando con caracteres extendidos, los flujos de **char** son los apropiados.

Debido a que la mayor parte de los flujos están basados en **char**, los nombres correspondientes a éstos se usarán en los ejemplos y los análisis en el resto de este capítulo y en todo el libro.

RECUERDE *En este capítulo y todo el libro, los nombres de flujo de **char**, como **ios** y **ofstream**, se usan en los ejemplos y los análisis.*

Flujos predefinidos de C++

Cuando un programa de C++ empieza a ejecutarse, se abren automáticamente cuatro flujos integrados. Son los siguientes:

Flujo	Significado	Dispositivo predeterminado
cin	Entrada estándar	Teclado
cout	Salida estándar	Pantalla
cerr	Salida de error estándar	Pantalla
clog	Versión para búfer de cerr	Pantalla

Los flujos **cout**, **clog** y **cerr** son instancias de **ostream**; el flujo **cin** es una instancia de **istream**. Por tanto, todos los flujos relacionados con **char** usan el encabezado **<iostream>**.

Como opción predeterminada, los flujos estándar se usan para comunicarse con la consola. Sin embargo, en entornos que dan soporte a redireccionamiento de E/S, los flujos estándar pueden redirigirse a otros dispositivos o archivos. Para mayor simplicidad, en los ejemplos de este capítulo se supone que no ha ocurrido ningún redireccionamiento de E/S.

El C++ estándar también define cuatro flujos adicionales: **win**, **wout**, **werr** y **wlog**. Son versiones de caracteres extendidos de los flujos estándar, y están basados en caracteres de tipo **wchar_t**. Los caracteres extendidos se usan para contener los conjuntos de caracteres largos asociados con algunos idiomas.

Las marcas de formato

Cada flujo está asociado con un conjunto de marcas de formato que controlan la manera en que la información se presenta. Estas marcas están contenidas en una enumeración de máscaras de bits llamada **fmtflags** que está definida por **ios_base**. Debido a que la formación es un tema muy amplio, se cubre de manera independiente en el capítulo 6. Por tanto, el análisis de las marcas de formato y las soluciones que las usan se pospondrá hasta entonces.

Los manipuladores de E/S

El sistema de E/S de C++ proporciona varios *manipuladores* que son funciones que pueden incluirse en una expresión de E/S formada. Se usan para establecer o limpiar las marcas de formato mencionadas en la sección anterior. También pueden usarse para otros fines, como salida a un carácter nulo o para omitir espacios en blanco en la entrada. Algunos manipuladores, como **endl** (que inserta una nueva línea en un flujo de salida), resultan familiares para todos los programadores de C++. Otros son menos conocidos. También es posible crear manipuladores propios.

Los manipuladores integrados se describen de manera detallada en el capítulo 6, donde se presentan soluciones relacionadas con la formación de datos. Sin embargo, en este capítulo se muestra cómo crear manipuladores propios. Los manipuladores personalizados pueden usarse para el fin que desee. Un uso común consiste en proporcionar un medio conveniente para controlar un dispositivo que no es estándar, como un graficador, que requiere códigos de formato o posicionamiento especiales.

Revisión de errores

La E/S de archivo plantea un desafío especial cuando se trata de manejo de errores, porque las fallas de E/S son una posibilidad real cuando se leen y escriben archivos. A pesar de que el hardware de computación (e Internet) es mucho más confiable que en el pasado, aún falla mucho, y cualquier falla debe manejarse de una manera consistente con las necesidades de su aplicación. En general, su código debe monitorear todas las operaciones de archivo en busca de errores y tomar la acción apropiada, si ocurre alguna.

El sistema de E/S de C++ proporciona amplias opciones para detección de errores. Como ya se mencionó, **ios_base** define un tipo llamado **iostate** que representa las diversas clases de errores que ocurren, codificados en una máscara de bits. Estas marcas de errores están definidas por los siguientes valores:

badbit	Establece si ha ocurrido un error catastrófico.
failbit	Establece si ha ocurrido un error del que es posible recuperarse.
eofbit	Establece si se ha alcanzado el final del archivo. (Ésta no es necesariamente una condición de error.)
goodbit	Un valor que indica que ninguno de los otros bits se ha establecido.

Observe que **eofbit** se incluye en la lista de marcas. Una condición de final de archivo no siempre representa un error. Esa determinación está basada en el contexto. (Por ejemplo, si está buscando a propósito el final del archivo, ¡no será un "error" que lo encuentre!) Recuerde que **ios_base** se hereda de **basic_ios**, de modo que esas marcas de formato son miembros de todas las clases de flujo. En el caso de flujos de **char**, por lo general se aludirá a esos valores mediante la especialización **ios** (por ejemplo, **ios::failbit**).

En la clase **basic_ios** están definidas varias funciones que pueden obtener el estado de las marcas de **iostate**. Aquí se muestran:

bool bad() const	Devuelve true si se establece **badbit**.
bool eof() const	Devuelve true si se establece **eofbit**.
bool fail() const	Devuelve true si se establece **failbit**.
bool good() const	Devuelve true si no se establecen bits.
iostate rdstate() const	Devuelve el valor de máscara de bits actual asociado con el flujo.

Puede usar estas funciones para buscar errores. Por ejemplo, una manera de confirmar que no han ocurrido errores consiste en llamar a **good()** en el flujo, como se muestra aquí:

```
if (miflujo good()) cout << "No hay errores. \n";
```

Otra manera de revisar errores consiste en usar la función **rdstate()**, que se muestra aquí:

iostate rdstate() const

Devuelve un valor en que están codificados los bits de estado. Por ejemplo, esta secuencia informa el éxito o la falla de una operación de E/S:

```
if(!(miflujo.rdstate() & (ios::badbit | ios::failbit))) {
  cout << "Archivo escrito correctamente. \n";
} else {
  cout << "Ha ocurrido un error de archivo.";
}
```

Por supuesto, por lo general es más fácil llamar simplemente a **good()**.

Una vez que se ha establecido un bit de error, permanece hasta que se limpia. Para limpiar un error, llame a **clear()**. Está definido por **ios** y se muestra aquí:

void clear(iostate *marca* = ios::goodbit)

Limpia (es decir, restablece) todas las marcas. Luego establece las marcas en *marca*. Puede establecer más de una marca al unirlas con el operador lógico OR. Como opción predeterminada, no se establecen marcas; por tanto, simplemente se limpian todas las condiciones de error.

También puede probar el estado de un flujo mediante el uso del operador **!**. Como ya se explicó, **!** devuelve la salida de **fail()**. Por tanto, si un flujo ha experimentado un error, entonces devolverá true. Por ejemplo,

```
if(!miflujo()) {
  // . . . ocurrió un error
}
```

Otra manera de manejar errores consiste en usar manejo de excepciones. Esta técnica se describe de manera detallada en la solución *Use excepciones para detectar y manejar errores de E/S.*

En los ejemplos de este capítulo, cualquier error de E/S que ocurra se manejará con el simple despliegue de un mensaje. Aunque es aceptable para los programas de ejemplo, por lo general en las aplicaciones reales será necesario proporcionar una respuesta más sofisticada a un error de E/S. Por ejemplo, tal vez quiera dar al usuario la capacidad de volver a probar la operación, especificar una operación alterna o manejar de otra manera el problema. La prevención de la pérdida o la corrupción de datos es uno de los principales objetivos. Para ser un estupendo programador es necesario saber cómo manejar de manera efectiva las cosas que podrían salir mal cuando falla una operación de E/S.

Un tema final: un error común cuando se manejan archivos consiste en olvidarse de cerrar un archivo cuando se ha dejado de usar. Los archivos abiertos usan recursos del sistema. Por tanto, hay límites para el número de archivos que pueden abrirse a la vez. El cierre de un archivo también asegura que cualquier dato escrito en el archivo realmente se escribe en el dispositivo físico. Por tanto, la regla es muy simple: Si abre un archivo, ciérrelo. Aunque los archivos se cierran automáticamente cuando se ejecuta el destructor de un flujo de archivo (como al final de una aplicación), es mejor no depender de esto porque puede llevar a hábitos descuidados e incorrectos. Es mejor cerrar explícitamente cada archivo cuando ya no se necesita, manejando cualquier error que podría ocurrir. Por esto, todos los archivos se cierran explícitamente en los ejemplos de este capítulo, aunque el programa se haya terminado.

Apertura y cierre de un archivo

Antes de que tenga lugar cualquier operación de E/S en un archivo, éste debe abrirse. Aunque los puntos específicos difieren de acuerdo con el tipo de archivo que se está abriendo, el procedimiento general es el mismo para todos los tipos. Por esto, tiene sentido describir las técnicas básicas de apertura de archivos en un solo lugar, en vez de hacerlo en cada solución. Para mayor conveniencia, en el siguiente análisis se usan los nombres definidos por las especializaciones de **char**, pero algunas técnicas básicas también se aplicarían a archivos de caracteres extendidos.

En C++, un archivo se abre al vincularlo con un flujo. Por tanto, antes de que pueda abrir un archivo, primero debe obtener una instancia de flujo. Hay tres tipos de flujo: entrada, salida y entrada/salida. Para crear un flujo de entrada de archivo, se usa **ifstream**. Para crear uno de salida, se usa **ofstream**. Los flujos que realizarán operaciones tanto de entrada como de salida se declaran como objetos de la clase **fstream**. Por ejemplo, este fragmento crea un flujo de entrada, uno de salida y uno capaz de entrada y salida:

```
ifstream entrada;  // entrada
ofstream salida;   // salida
fstream es;        // entrada y salida
```

Una vez que ha creado un flujo, puede asociarlo con un archivo al usar **open()**. Esta función es un miembro de cada una de las tres clases de flujo. A continuación se muestra el prototipo para cada uno:

void ifstream::open(const char **nombrear*, ios::openmode *modo* = ios::in)

void ofstream::open(const char **nombrear*, ios::openmode *modo* = ios::out)

void fstream::open(const char **nombrear*, ios::openmode *modo* = ios::in | ios::out)

Aquí, *nombrear* es el nombre del archivo; puede incluir un especificador de ruta. El valor de *modo* determina la manera en que se abre un archivo. Debe ser uno o más de los valores definidos por **openmode**, que es una enumeración definida por **ios** (mediante su clase de base **ios_base**). He aquí los valores definidos por **openmode**:

app	La salida se adjunta al final del archivo.
ate	Se hace una búsqueda inicial al final del archivo.
binary	El archivo se abre en modo binario en lugar de texto. (El modo de texto es la opción predeterminada.)
in	El archivo se abre para entrada. (No puede usarse con **ofstream**.)
out	El archivo se abre para salida. (No puede usarse con **ifstream**.)
trunc	El archivo se trunca.

Puede incluirse más de un valor de modo al usar el operador OR junto construcción **|**. A continuación se realiza una descripción detallada de su efecto.

El valor **in** especifica que el archivo puede contener entrada. El valor **out**, que puede contener salida. En todos los casos, por lo menos debe usarse uno de estos valores cuando se abre un archivo.

La inclusión de **app** causa que toda la salida al archivo se adjunte al final. Estos valores sólo pueden usarse con archivos con capacidad de salida. La inclusión de **ate** causa una búsqueda al final del archivo cuando se abre éste. A pesar de este comportamiento de **ate**, las operaciones de E/S aun pueden ocurrir en cualquier lugar dentro del archivo.

El valor **binary** causa que un archivo se abra en modo binario. Como opción predeterminada, todos los archivos se abren en modo de texto. En este modo, pueden darse varias traducciones de carácter; por ejemplo, es posible que la secuencia retorno de carro/avance de línea se convierta en nueva línea. Sin embargo, cuando un archivo se abre en modo binario, no ocurrirá esta traducción de caracteres. Es necesario comprender que cualquier archivo, sin importar si contiene texto

formado o datos sin trabajar, puede abrirse en modo binario o de texto. La única diferencia es si tienen lugar las traducciones de caracteres.

El valor **trunc** causa que se destruya el contenido de un archivo preexistente del mismo nombre, y el archivo se trunca a una longitud cero.

Debido a que **ios** hereda **ios_base**, a menudo verá estos valores de modo calificados con **ios::** en lugar de **ios_base::**. Por ejemplo, a menudo verá **ios::out** en lugar de **ios_base::out**. En este libro se utiliza la forma **ios::** porque es más corta. (En realidad, también podría usar constructores como **ofstream::out** o **ifstream::in**, pero lo tradicional es que se use **ios::**.)

Para unir los diversos elementos, el siguiente fragmento crea un flujo de salida llamado **archsalida** y usa **open()** para vincularlo con un archivo llamado **prueba.dat**. Aunque usa **ofstream** (que crea un flujo de archivo de salida), el método general se aplica a todos los flujos de archivo.

```
// Crea un objeto de ofstream.
ofstream archsalida;

// Abre un archivo en archsalida
archsalida.open("prueba.dat");
```

Esta secuencia crea primero un objeto de **ofstream** llamado **archsalida**, que no está vinculado con un archivo. Por tanto, aunque **archsalida** sea una instancia de **ofstream**, no puede usarse para escribir salida porque no está asociada aún con un archivo específico. La llamada a **open()** vincula **archsalida** con el archivo llamado **prueba.dat** y abre el archivo para operaciones de salida. Después de que regresa **open()**, es posible escribir en un archivo mediante **archsalida**. Debido a que el parámetro de modo de **open()** tiene como opción predeterminada automática **ios_out**, no es necesario especificarlo explícitamente en este caso.

Aunque no hay nada incorrecto con el método anterior de "dos pasos", todas las clases de flujo de archivo (**fstream**, **ofstream** e **ifstream**) le permiten abrir un archivo al mismo tiempo que el objeto de flujo se crea al pasar el nombre del archivo al constructor. He aquí los constructores de flujo que le permiten especificar un archivo:

ofstream(const char **nombrear*, ios::openmode *modo* = ios::out)

ifstream(const char **nombrear*, ios::openmode *modo* = ios::in)

fstream(const char **nombrear*, ios::openmode *modo* = ios::in | ios::out)

Como puede ver, el parámetro *modo* tiene como opción predeterminada un valor apropiado para el flujo. Por ejemplo, he aquí una manera mucho más compleja de crear **archsalida** y vincularla con **prueba.dat**:

```
ofstream archsalida("prueba.dat");
```

Cuando se ejecuta esta instrucción, se construye un objeto de **ofstream** que se vincula con un archivo llamado **prueba.dat**, y luego se abre ese archivo para salida. Como antes, aunque **ofstream** se usa en este ejemplo, el mismo método general se aplica a todos los flujos de archivo.

Es importante comprender que **open()** y los constructores de flujo de archivo *tratan* de abrir un archivo. Sin embargo, este intento puede fallar por varias razones, como cuando el llamador no tiene los permisos apropiados de seguridad para abrir el archivo, o cuando se alcanza el límite de archivos abiertos al que da soporte el entorno. Por tanto, antes de usar un archivo, debe confirmar que se ha abierto correctamente. Hay varias maneras de hacer esto. Una consiste en llamar a **is_open()** en la instancia de flujo de archivo. Aquí se muestra:

bool is_open()

Devuelve true si el archivo está abierto y false, si no. Por ejemplo, la siguiente secuencia verifica que **archsalida** en realidad está abierto:

```
ofstream archsalida("prueba.dat");
// Verifica que el archivo se ha abierto correctamente.
if(!archsalida.is_open()) {
  cout << "no pudo abrirse archsalida. \n";
  // maneja el error
}
```

Esto funciona porque si falla el intento de abrir el archivo, entonces **is_open()** devuelve false, porque **archsalida** no está abierto. Es importante comprender que puede usar **is_open()** en cualquier momento en que necesite saber si un archivo está abierto. Su uso no está limitado a verificar que la operación de apertura fue correcta.

Aunque el uso de **is_open()** es válido, y se aplica de manera ocasional en los ejemplos de este libro, hay otros modos de verificar que el archivo se ha abierto de forma correcta. Estas otras maneras se basan en el hecho de que la falla al abrir crea una condición de error en el flujo. En forma específica, si no es posible abrir un archivo (mediante una llamada explícita a **open()** o mediante el constructor de flujo de archivo), entonces la marca de falla **failbit** se establecerá en el flujo para indicar una falla de E/S. Por tanto, si no puede abrirse, una llamada a **fail()** en el flujo devolverá true. Esto significa que puede detectar una falla al llamar a **fail()** en el flujo. Por tanto, he aquí otro modo de detectar una falla en la apertura:

```
ofstream archsalida("prueba.dat");
if(archsalida.fail()) {
  cout << "no pudo abrirse archsalida. \n";
  // maneja el error
}
```

En este caso, si falla el intento de abrir el archivo, **fail()** devolverá true. Sin embargo, he aquí una manera más simple.

Como se explicó antes, cuando el operador **!** se aplica a un flujo de archivo, devuelve el resultado de **fail()** llamado en el mismo flujo. Por tanto, para probar una falla en la apertura, puede usar esta secuencia:

```
ofstream archsalida("prueba.dat");
if(!archsalida) {
  cout << "no pudo abrirse archsalida. \n";
  // maneja el error
}
```

Ésta es la forma que verá con frecuencia en código escrito de manera profesional.

Cuando haya terminado con un archivo, debe asegurarse de que está cerrado. En general, un archivo se cierra automáticamente con el destructor del flujo de archivo cuando las instancias de éste salen del ámbito, como cuando termina un programa. También puede cerrar explícitamente un archivo al llamar a **close()**, que tiene soporte en todas las clases de flujo de archivo. Aquí se muestra:

void close()

El cierre de un archivo causa que el contenido de cualquier búfer se limpie y se liberen los recursos del sistema vinculados con el archivo.

Aunque los archivos se cierran automáticamente cuando se destruye el flujo de archivo, muchos programadores creen que es una mejor práctica cerrarlos explícitamente cuando ya no se necesiten. Una razón para esto es que la apertura de archivos consume recursos del sistema. El cierre de archivos libera estos recursos. Por tanto, en todos los ejemplos de este capítulo se cierran explícitamente todos los archivos, aun al final de un programa, simplemente para ejemplificar de manera explícita el uso de **close()** y para destacar que los archivos deben cerrarse.

Escriba datos formados en un archivo de texto

Componentes clave		
Encabezados	**Clases**	**Funciones**
<fstream>	ofstream	void close() bool good() const void open(const char **nombrear*, ios::openmode *modo* = ios::out)
<ostream>		<<

C++ le ofrece dos maneras de escribir datos en un archivo. En primer lugar, puede escribir datos sin formato en su forma simple, binaria. En segundo lugar, puede escribir datos formados. Éstos son datos en su forma de texto, legible para los seres humanos. En este método, el formato de los datos escritos en el archivo será el mismo que vería en la pantalla. A un archivo que contiene datos formados suele denominársele *archivo de texto*. La escritura de datos formados en un archivo de texto es el tema de esta solución.

Paso a paso

Para escribir datos formados en un archivo se requieren estos pasos:

1. Cree una instancia de **ofstream**.
2. Abra el archivo al llamar a **open()** en la instancia de **ofstream** creada en el paso 1. Como opción, puede abrir el archivo al mismo tiempo que crea el objeto de **ofstream**. (Consulte la sección *Análisis* de esta solución.)
3. Confirme que el archivo se ha abierto correctamente.
4. Escriba datos en el archivo al usar el operador de inserción **<<**.
5. Cierre el archivo al llamar a **close()**.
6. Confirme que las operaciones de escritura han sido correctas. Esto puede hacerse al llamar a **good()** en el flujo de salida.

Análisis

Una revisión general de la apertura y el cierre de un archivo se encuentra en *Apertura y cierre de un archivo*, casi al principio de este capítulo. Aquí se presentan los detalles específicos relacionados con **ofstream**.

Para crear un flujo de salida vinculado con un archivo, cree un objeto de tipo **ofstream**. Tiene estos dos constructores:

ofstream()

explicit ofstream(const char **nombrear*, ios::openmode *modo* = ios:out)

El primero crea una instancia de **ofstream** que no está vinculada con un archivo. El segundo crea una instancia de **ofstream** y luego abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que *modo* tiene como opción predeterminada **ios::out**. Esto causa que se cree el archivo, y que se destruya cualquier archivo anterior con el mismo nombre. Además, el archivo se abre automáticamente para salida de texto. (Como opción predeterminada, todos los archivos se abren en modo de texto. Para el caso de la salida binaria, debe solicitar explícitamente el modo binario.) La clase **ofstream** requiere el encabezado **<fstream>**.

Si utiliza el constructor predeterminado, entonces necesitará vincular un archivo con la instancia de **ofstream** después de construirla. Para esto, llame a **open()**. Aquí se muestra la versión definida por **ofstream**:

void open(const char **nombrear*, ios::openmode *modo* = ios::out)

Abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que, al igual que el constructor **ofstream**, la opción predeterminada de *modo* es **ios::out**.

Antes de escribir en el archivo, debe confirmar que se ha abierto. Puede hacer esto de diversas maneras. El método usado en esta solución consiste en aplicar el operador **!** a la instancia de **ofstream**. Recuerde que el operador **!** devuelve la salida de una llamada a **fail()** en el flujo. Por tanto, si devuelve true, la operación de apertura ha fallado.

Una vez que se ha abierto correctamente un archivo de salida, puede escribir salida formada en él mediante el operador de inserción **<<**. Está definido por todos los objetos de tipo **ostream**, entre ellos **ofstream**, porque hereda **ostream**. Utiliza el encabezado **<ostream>**, que suele incluirse con **<fstream>**, de modo que no necesitará incluirlo explícitamente. El operador **<<** se usa para escribir salida formada a un archivo, de la misma manera que se usa para escribir salida en la consola mediante **cout**. Por ejemplo, suponiendo que **archsalida** representa un archivo de salida abierto, la siguiente instrucción escribe un entero, una cadena y un punto flotante en él:

```
archsalida << 10 << " Esto es una prueba " << 1.109;
```

Debido a que el archivo se ha abierto para salida de texto, esta información se escribe en su forma legible para los seres humanos. Por tanto, el archivo contendrá lo siguiente:

```
10 Esto es una prueba 1.109
```

Cuando haya terminado de escribir en un archivo, debe cerrarlo. Esto se hace al llamar a **close(),** que se muestra aquí:

void close()

El archivo se cierra automáticamente cuando se llama al destructor de **ofstream**. Sin embargo, por las razones establecidas en *Apertura y cierre de un archivo*, en este libro se llamará explícitamente a **close()** en todos los casos.

En esta solución se comprueba que no han ocurrido errores de E/S al llamar a **good()** en el flujo. Aquí se muestra:

bool good() const

Devuelve true si no están establecidas marcas de error.

Ejemplo

En el siguiente ejemplo se escriben datos formados en un archivo de texto llamado **prueba.dat**. Observe que no está especificado el parámetro *modo* del constructor de **ofstream**. Esto significa que la opción predeterminada es **ios::out**. Para este ejemplo, en el programa se utiliza la función **good()** para informar del éxito o la falla de las operaciones de archivo. Como se explicó, también son posibles otros métodos.

```
// Escribe salida formada en un archivo de texto.

#include <iostream>
#include <fstream>

using namespace std;

int main()
{
  // Crea un objeto de ofstream y trata de abrir
  // el archivo prueba.dat.
  ofstream archsalida("prueba.dat");

  // Verifica que el archivo se abrió correctamente.
  if(!archsalida) {
    cout << "No se puede abrir el archivo.\n";
    return 1;
  }

  // Escribe salida en el archivo.
  archsalida << 10 << " " << -20 << " " << 30.2 << "\n";
  archsalida << "Esto es una prueba.";

  // Cierra explícitamente el archivo.
  archsalida.close();

  if(!archsalida.good()) {
    cout << "Ha ocurrido un error con el archivo.";
    return 1;
  }
}
```

Aquí se muestra el contenido de **prueba.dat**:

```
10 -20 30.2
Esto es una prueba.
```

Como se observa, los datos están almacenados en formato de texto, legible para el ser humano.

Opciones

Cuando se usa **ofstream**, el parámetro *modo* de **open()** o el constructor de **ofstream** debe incluir **ios::out** (que es la opción predeterminada), pero también puede incluir otros valores. Uno de los más útiles es **ios::app**, porque causa que toda la salida se presente al final del archivo. Esto significa que no se perderá el contenido de un archivo preexistente del mismo nombre. En cambio, la salida se agrega al final del contenido anterior. Por ejemplo, si utiliza esta llamada a **ofstream()** para abrir **prueba.dat** en el programa anterior:

```
ofstream archsalida("prueba.dat", ios::out | ios::app);
```

entonces la salida se escribirá al final del archivo. Por tanto, cada vez que ejecute el programa, el archivo se hará más largo.

Para provocar una búsqueda inicial al final del archivo, incluya **ios::ate**. Después de la búsqueda inicial al final, la salida puede darse en cualquier lugar.

Aunque el uso de **good()** es una manera conveniente de confirmar el éxito de una operación de salida con formato, no es la única manera. Por ejemplo, puede usar las funciones **bad()** o **fail()**. También puede usar **rdstate()**. Consulte *Revisión de errores* en la revisión general, para conocer más detalles.

Otra manera de revisar posibles errores de E/S es mediante el uso de excepciones. Para ello, debe especificar los errores que lanzarán excepciones al llamar a **exceptions()** en el objeto **ofstream**. Luego debe capturar excepciones de tipo **ios_base::failure**. (Consulte *Use excepciones para detectar y manejar errores de E/S* para conocer más detalles.)

Si quiere escribir datos binarios, abra el flujo de salida en modo binario. (Consulte *Escriba datos binarios sin formato en un archivo* para conocer más detalles.) Para leer datos formados de un archivo, utilice **ifstream**. (Consulte *Lea datos formados de un archivo de texto*.) Para abrir un archivo para entrada y salida, cree un objeto de **fstream**. (Consulte *Lea un archivo y escriba en él*.)

Lea datos formados de un archivo de texto

Componentes clave		
Encabezados	**Clases**	**Funciones**
<fstream>	ifstream	void close() bool good() const void open(const char **nombrear*, ios::openmode *modo* = ios::in)
<istream>		>>

Puede leer datos formados de un archivo de texto al usar las opciones de entrada formadas del sistema de E/S de C++. Aquí, *datos formados* significa la forma de texto, legible para los seres humanos de los datos, en lugar de su representación binaria sin trabajar. Por ejemplo, dado un archivo que contiene lo siguiente:

10 Hola 123.23

puede usar las características de entrada formada de C++ para leer el entero 10, la cadena Hola y el valor de punto flotante 123.23, almacenando el resultado en un valor **int**, **string** y **double**, respectivamente. En general, puede leer valores de cadena, enteros, booleanos y de punto flotante que están almacenados en su forma legible para los seres humanos. En esta solución se muestra este proceso.

Paso a paso

Para leer datos formados de un archivo se requieren estos pasos:

1. Cree una instancia de **ifstream**.
2. Abra el archivo al llamar a **open()** en la instancia de **ifstream** creada en el paso 1. Como opción, puede abrir el archivo al mismo tiempo que crea el objeto de **ifstream**. (Consulte la sección *Análisis* de esta solución.)
3. Confirme que el archivo se ha abierto correctamente.
4. Lea datos del archivo al usar el operador de extracción >>.
5. Cierre el archivo al llamar a **close()**.
6. Confirme que las operaciones de lectura han sido correctas. Esto puede hacerse al llamar a **good()** en el flujo de entrada.

Análisis

Una revisión general de la apertura y el cierre de un archivo se encuentra en *Apertura y cierre de un archivo*, casi al principio de este capítulo. Aquí se presentan los detalles específicos relacionados con **ifstream**.

Para crear un flujo de entrada vinculado con un archivo, cree un objeto de tipo **ifstream**. Tiene estos dos constructores:

ifstream()

explicit ifstream(const char **nombrear*, ios::openmode *modo* = ios:in)

El primero crea una instancia de **ifstream** que no está vinculada con un archivo. El segundo crea una instancia de **ifstream** y luego abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que *modo* tiene como opción predeterminada **ios::in**. Esto causa que el archivo se abra automáticamente para entrada de texto. (Como opción predeterminada, todos los archivos se abren en modo de texto. Para el caso de la entrada binaria, debe solicitar explícitamente el modo binario.) Es obligatorio que exista el archivo especificado por *nombrear*. De lo contrario, se producirá un error. La clase **ifstream** requiere el encabezado **<fstream>**.

Si utiliza el constructor predeterminado, entonces necesitará vincular un archivo con la instancia de **ifstream** después de construirla. Para esto, llame a **open()**. Aquí se muestra la versión definida por **ifstream**:

void open(const char **nombrear*, ios::openmode *modo* = ios::in)

Abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que, al igual que el constructor **ifstream**, la opción predeterminada de *modo* es **ios::in**.

Antes de tratar de leer del archivo, debe confirmar que se ha abierto. Puede hacer esto de diversas maneras. El método usado en esta solución consiste en aplicar el operador **!** a la instancia de **ifstream**. Recuerde que el operador **!** devuelve la salida de una llamada a **fail()** en el flujo. Por tanto, si devuelve true, la operación de apertura ha fallado.

Una vez que se ha abierto correctamente un archivo de entrada, puede leer datos formados de él mediante el operador de extracción **>>**. Está definido por todos los objetos de tipo **istream**, entre ellos **ifstream**, porque hereda **istream**. Utiliza el encabezado **<istream>**, que suele incluirse con **<fstream>**, de modo que no necesitará incluirlo explícitamente. El operador **>>** se usa para leer entrada formada de un archivo, de la misma manera que se utiliza para leer entrada de la consola mediante **cin**. Por ejemplo, suponiendo que **archentrada** representa un archivo de entrada abierto, la siguiente instrucción lee un valor **int**, **string** y **double** de él:

```
int x;
string cad;
double val;

archentrada >> x;
archentrada >> cad;
archentrada >> val;
```

Suponiendo que el archivo al que se hace referencia con **archentrada** contiene lo siguiente:

10 Hola 123.23

entonces, después de leer los datos, **x** contendrá el valor 10, **cad** contendrá la cadena Hola y **val** contendrá el valor 123.23.

Cuando haya terminado de leer un archivo, debe cerrarlo. Esto se hace al llamar a **close(),** que se muestra aquí:

void close()

El archivo se cierra automáticamente cuando se llama al destructor de **ifstream**. Sin embargo, por las razones establecidas en *Apertura y cierre de un archivo*, en este libro se llamará explícitamente a **close()** en todos los casos.

En esta solución se comprueba que no han ocurrido errores de E/S al llamar a **good()** en el flujo. Aquí se muestra:

bool good() const

Devuelve true si no están establecidas marcas.

Ejemplo

En el siguiente ejemplo se muestra cómo leer entrada formada de un archivo de texto. Se lee el archivo producido por el programa de ejemplo de *Escriba datos formados en un archivo de texto.*

```
// Lee datos formados de un archivo.
//
//   Nota: este programa lee el archivo prueba.dat
//   producido por el programa de ejemplo de
//
//        Escriba datos formados en un archivo de texto
//
//   El archivo prueba.dat creado por ese programa
//   contiene los siguientes datos:
//
//     10 -20 30.2
```

```
//     Esto es una prueba.

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
  int i, n;
  double d;
  string cad;

  // Crea un objeto de ifstream y trata de abrir el archivo prueba.dat.
  ifstream archentrada("prueba.dat");

  // Verifica que el archivo se ha abierto correctamente.
  if(!archentrada) {
    cout << "No se puede abrir el archivo.\n";
    return 1;
  }

  // Lee los datos formados.
  archentrada >> i;
  archentrada >> n;
  archentrada >> d;
  archentrada >> cad;

  // Cierra el archivo de entrada.
  archentrada.close();

  // Confirma que no ocurrieron errores en la entrada.
  if(!archentrada.good()) {
    cout << "Ha ocurrido un error con el archivo.";
    return 1;
  }

  // Despliega los datos.
  cout << i << " " << n << " " <<
          d << " " << cad << "\n";

  return 0;
}
```

Aquí se muestra la salida:

```
10 -20 30.2 Esto
```

Observe que sólo se despliega la palabra "Esto" en lugar de toda la frase "Esto es una prueba". Se debe a que el operador >> utiliza el espacio en blanco como un separador de campo. Por tanto, la línea

```
archentrada >> cad;
```

deja de leer caracteres cuando se encuentra el primer espacio, que es el que sigue a "Esto" en la frase. Se requieren operaciones de entrada adicionales para leer el resto de la frase.

Opciones

Como ya se señaló, cuando se lee una cadena, el operador **>>** lee caracteres hasta que se encuentra un espacio en blanco. Si quiere leer una línea de texto completo, entonces querrá usar una de las funciones de entrada no formada, como **getline()**. Consulte *Use **get()** y **getline()** para leer un archivo.*

En algunas situaciones de entrada, querrá leer datos hasta que alcance el final del archivo. Puede determinar cuando se ha encontrado el final de un archivo al llamar a **eof()** en el flujo. Consulte *Detección de EOF*.

Aunque el uso de **good()** es una manera conveniente de confirmar el éxito de una operación de entrada con formato, no es la única manera. Por ejemplo, puede usar las funciones **bad()** o **fail()**. También puede usar **rdstate()** o el operador **!** en el flujo. Consulte *Revisión de errores* en la revisión general, para conocer más detalles. También puede revisar posibles errores de E/S mediante el uso de excepciones. Para ello, debe especificar los errores que lanzarán excepciones al llamar a **exceptions()** en el objeto **ifstream**. Luego debe capturar excepciones de tipo **ios_base::failure**. (Consulte *Use excepciones para detectar y manejar errores de E/S* para conocer más detalles.)

Cuando se lee una cadena mediante el operador de extracción **>>**, debe evitarse el uso de una matriz de caracteres para recibir la entrada. Use, en cambio, una **string**. Si utiliza una matriz de caracteres, entonces es posible que el final de la matriz pueda desbordarse con una secuencia de entrada inesperadamente larga. Ésta es una fuente de la famosa falla de seguridad "desbordamiento de búfer". Debido a que **string** es una estructura de datos dinámica, puede tratar mejor con una entrada inesperadamente larga. En algunos casos, podría ser aún mejor evitar por completo el uso de **>>** para leer cadenas, dependiendo, en cambio, de las funciones de entrada sin formato. Consulte *Lea datos binarios sin formato de un archivo*.

Si quiere leer datos binarios, abra el flujo de entrada en modo binario. (Consulte *Lea datos binarios sin formato de un archivo* para conocer más detalles.) Para escribir datos formados en un archivo, utilice **ofstream**. (Consulte *Escriba datos formados en un archivo de texto*.) Para abrir un archivo para entrada y salida, cree un objeto de **fstream**. (Consulte *Lea un archivo y escriba en él*.)

Escriba datos binarios sin formar en un archivo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<fstream>	ofstream	void close() bool good() const void open(const char **nombrear*, ios::openmode *modo* = ios::out) ostream &write(const char **cad*, streamsize *num*)

En la solución *Escriba datos formados en un archivo* se describió cómo escribir datos formados (es decir, basados en texto) en un archivo de texto. Aunque este tipo de salida es útil en muchas situaciones, a menudo querrá escribir datos sin formato. Aquí "sin formato" significa que están escritos byte por byte en su forma binaria sin trabajar, sin traducción o sin formato con una representación legible para los seres humanos. La salida no formada suele usarse para crear archivos de datos,

en que éstos están almacenados en su forma binaria. Por supuesto, también puede utilizar salida no formada para crear un archivo de texto al escribir valores tipo **char**. Sin importar cuál sea el propósito, en esta solución se muestra el procedimiento básico empleado para escribir salida sin formato en un archivo.

Paso a paso

Una manera de escribir datos no formados en un archivo requiere estos pasos:

1. Cree una instancia de **ofstream**.
2. Abra el archivo al llamar a **open()** en la instancia de **ofstream** creada en el paso 1. Como opción, puede abrir el archivo al mismo tiempo que crea el objeto de **ofstream**. (Consulte la sección *Análisis* de esta solución.)
3. Confirme que el archivo se ha abierto correctamente.
4. Una manera de escribir datos no formados en el archivo consiste en llamar a **write()**.
5. Cierre el archivo al llamar a **close()**.
6. Confirme que las operaciones de escritura han sido correctas. Esto puede hacerse al llamar a **good()** en el flujo de entrada.

Análisis

Una revisión general de la apertura y el cierre de un archivo se encuentra en *Apertura y cierre de un archivo*, casi al principio de este capítulo. Aquí se presentan los detalles específicos relacionados con el uso de **ofstream** para escribir datos binarios, sin formato.

Para realizar salida binaria sin formato, debe tener un objeto de tipo **ofstream** que dé soporte a operaciones binarias. La clase **ofstream** usa el encabezado **<fstream>** y define estos dos constructores:

ofstream()

explicit ofstream(const char **nombrear*, ios::openmode *modo* = ios:out)

El primero crea una instancia de **ofstream** que no está vinculada aún con un archivo. El segundo crea una instancia de **ofstream** y luego abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que *modo* tiene como opción predeterminada **ios::out**, pero no incluye la marca **ios::binary**. Como opción predeterminada, un archivo se abre en modo de texto. Para abrirlo para salida sin formato, el argumento de modo debe especificar **ios::out** y **ios::binary**. Por ejemplo, lo siguiente abre un archivo llamado **prueba.dat** para salida binaria:

```
ofstream archsalida("prueba.dat", ios::out | ios::binary);
```

Esto causa que se destruya cualquier archivo anterior con el nombre **prueba.dat** y que se cree un nuevo archivo.

Cuando se especifica la marca de modo **binary**, los datos se escriben en su forma binaria, sin trabajar, con lo que se evitan posibles traducciones de caracteres (como la conversión de nueva línea en la secuencia retorno de carro/avance de línea) que podría ocurrir cuando el archivo se abre en modo de texto. (Recuerde que si no se especifica **ios::binary**, el archivo se abre de manera automática en modo de texto.) Si no se usa el modo binario, puede ocurrir que el patrón de bits contenido en el archivo sea diferente del que se encuentra en el bloque original de la memoria. Por tanto, siempre debe especificar **ios::binary** cuando abra un archivo para salida binaria.

Si utiliza el constructor predeterminado, entonces necesitará vincular un archivo con la instancia de **ofstream** después de construirla. Para esto, llame a **open()**. Aquí se muestra la versión definida por **ofstream:**

void open(const char **nombrear*, ios::openmode *modo* = ios::out)

Abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que, al igual que el constructor **ofstream**, la opción predeterminada de *modo* es **ios::out**. Por tanto, debe especificar explícitamente **ios::out** y **ios::binary** para escribir datos binarios sin formato.

Antes de tratar de escribir en el archivo, debe confirmar que se ha abierto. Puede hacer esto de diversas maneras. El método usado en esta solución consiste en aplicar el operador **!** a la instancia de **ofstream**. Recuerde que el operador **!** devuelve la salida de una llamada a **fail()** en el flujo. Por tanto, si devuelve true, la operación de apertura ha fallado.

Una manera de escribir salida sin formato a un archivo consiste en usar la función **write()**. Escribe un bloque de datos en un flujo. Aquí se muestra:

ostream &write(const char **buf*, streamsize *num*)

Aquí *buf* es un apuntador al bloque de memoria que se escribirá y *num* especifica el número de bytes que se habrá de escribir. El tipo **streamsize** está definido como alguna forma de entero que puede contener el número más grande de bytes que es posible transferir en cualquier operación de E/S. La función devuelve una referencia al flujo. Aunque *buf* está especificado como **char** *, puede usar **write()** para escribir cualquier tipo de datos binarios. Simplemente convierta un apuntador a los datos a **char** * y especifique la longitud del bloque en bytes. (Recuerde que, en C++, un **char** es siempre exactamente de un byte de largo.) Por ejemplo, esta secuencia escribe en **archsalida** el valor **double** en **val**:

```
double val = 10.34;
archsalida.write((char *) &val, sizeof(double);
```

Debe comprender que los datos se escriben en su formato interno, de punto flotante. Por tanto, el archivo contiene la imagen de patrón de bits de **val**, no su forma legible para los seres humanos.

Cuando haya terminado de escribir en un archivo, debe cerrarlo. Esto se hace al llamar a **close(),** que se muestra aquí:

void close()

El archivo se cierra automáticamente cuando se llama al destructor de **ofstream**. Sin embargo, por las razones establecidas en *Apertura y cierre de un archivo*, en este libro se llamará explícitamente a **close()** en todos los casos.

En esta solución se comprueba que no han ocurrido errores de E/S al llamar a **good()** en el flujo. Aquí se muestra:

bool good() const

Devuelve true si no están establecidas marcas de error.

Ejemplo

En el siguiente ejemplo se demuestra la escritura de datos binarios en un archivo. Se crea una estructura llamada **inventario** que almacena el nombre, la cantidad y el costo de un artículo del

inventario. Luego, se crea una matriz de tres elementos de estructuras de **inventario** llamada **inv** y se almacena información de inventario en esa matriz. Luego escribe esa matriz en un archivo llamado **InvDat.dat**. Después de que termina el programa, el archivo contendrá una copia byte por byte de la información de **inv**.

```
// Usa write() para dar salida a un bloque de datos binarios.

#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

// Una estructura simple de inventario.
struct inventario {
  char producto[20];
  int cantidad;
  double costo;
};

int main()
{
  // Crea y abre un archivo para salida binaria.
  ofstream archsalida("InvDat.dat", ios::out | ios::binary);

  // Confirma que el archivo se abrió sin error.
  if(!archsalida) {
    cout << "No se puede abrir el archivo.\n";
    return 1;
  }

  // Crea algunos datos de inventario.
  inventario inv[3];

  strcpy(inv[0].producto,"Martillos");
  inv[0].cantidad = 3;
  inv[0].costo = 9.99;

  strcpy(inv[1].producto, "Pinzas");
  inv[1].cantidad = 12;
  inv[1].costo = 7.85;

  strcpy(inv[2].producto, "Llaves");
  inv[2].cantidad = 19;
  inv[2].costo = 2.75;

  // Escribe datos de inventario en el archivo.
  for(int i=0; i<3; i++)
    archsalida.write((const char *) &inv[i], sizeof(inventario));

  // Cierra el archivo.
  archsalida.close();
```

```
  // Confirma que no hubo errores de archivo.
  if(!archsalida.good()) {
    cout << "Ha ocurrido un error con el archivo.";
    return 1;
  }

  return 0;
}
```

Opciones

Otra manera de escribir datos sin formato en un flujo es llamando a **put()**. Aquí se muestra:

ostream &put(char *car*)

Esta función escribe el valor de bytes pasado en *car* al flujo asociado. (Recuerde que en C++, un **char** tiene un byte de largo. Por tanto, cada llamada a **put()** escribe un byte de datos.) La función devuelve una referencia al flujo. He aquí un ejemplo de la manera en que puede usarse. Suponiendo que **archsalida** es un flujo de salida abierto, lo siguiente escribe los caracteres en la cadena señalada por **cad**:

```
const char *cad = "Hola";
while(*cad) archsalida.put(*cad++);
```

Después de que se ejecuta la secuencia, el archivo contendrá los caracteres "Hola".

Tanto **put()** como **write()** pueden usarse en un flujo de salida de texto (es decir, un flujo no especificado como binario). Sin embargo, si lo hace, entonces pueden ocurrir algunas traducciones de carácter. Por ejemplo, una nueva línea se convertirá en una secuencia retorno de carro/avance de línea. En general, si está usando **put()** o **write()**, normalmente abrirá el archivo en operaciones binarias.

Como opción predeterminada, cuando se abre un archivo para salida, se destruye el contenido de cualquier archivo preexistente del mismo nombre. Puede evitar esto al incluir la marca **ios::app** en el parámetro de *modo* de **open()** o el constructor **ofstream**. Hace que toda la salida ocurra al final del archivo, preservando así su contenido. Para provocar una búsqueda inicial al final del archivo, incluya **ios::ate**. Después de la búsqueda inicial al final, la salida puede darse en cualquier lugar.

Aunque el uso de **good()** es una manera conveniente de confirmar el éxito de una operación de salida sin formato, no es la única manera. Por ejemplo, puede usar las funciones **bad()** o **fail()**. También puede usar **rdstate()**. Consulte *Revisión de errores* en la revisión general, para conocer más detalles. También puede revisar posibles errores de E/S mediante el uso de excepciones. Para ello, debe especificar los errores que lanzarán excepciones al llamar a **exceptions()** en el objeto **ofstream**. Luego debe capturar excepciones de tipo **ios_base::failure**. (Consulte *Use excepciones para detectar y manejar errores de E/S* para conocer más detalles.)

Otra manera de revisar errores cuando usa **write()** o **put()** consiste en revisar el estado del flujo. Debido a que **write()** o **put()** devuelve una referencia al flujo en que operan, puede aplicarse el operador **!** al objeto devuelto. Recuerde que cuando **!** se aplica a un flujo, devuelve el resultado de **fail()** aplicado al mismo flujo. Por tanto, puede probar si se hizo una llamada correcta a **write()** como ésta:

```
if(!write(...)) { // ... maneja el error de escritura
```

Por ejemplo, en el programa anterior, puede usar la siguiente secuencia para escribir los registros de inventario, confirmando el éxito de cada operación de salida en el proceso:

```
if(!write((const char *) &inv[i]), sizeof(inventario))) {
    cout << "Error al escribir el archivo.";
     // maneja el error ...
}
```

El hecho de tomar este método para revisar errores afina su código fuente. Sin embargo, debido a que cada llamada a **write()** también da como resultado que se evalúe una instrucción **if** (lo que toma tiempo), *no* afina el rendimiento de su programa. Como regla general, las excepciones ofrecen una mejor opción en este tipo de situación.

Para leer información binaria, sin formato de un archivo, consulte *Lea datos binarios sin formar de un archivo.* Para leer datos formados de un archivo, use **ifstream**. (Consulte *Lea datos formados de un archivo de texto.*)

Lea datos binarios sin formar de un archivo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<fstream>	ifstream	void close() bool good() const void open(const char **nombrear*, ios::openmode *modo* = ios::in) ostream &read(char **cad*, streamsize *num*)

En la solución *Lea datos formados de un archivo* se describió cómo leer datos formados (es decir, basados en texto) de un archivo de texto. Aunque este tipo de entrada es útil en muchas situaciones, a menudo querrá leer datos sin formato, en su forma binaria sin trabajar, sin ninguna traducción de caracteres (lo que es posible con la entrada formada). Por ejemplo, si fuera a crear una utilería de comparación de archivos, querría operar sobre los datos binarios dentro de los archivos, byte por byte. Sin importar cuál sea la necesidad, en esta solución se muestra el procedimiento básico empleado para leer entrada sin formato de un archivo.

Paso a paso

Una manera de leer datos no formados de un archivo requiere estos pasos:

1. Cree una instancia de **ifstream**.
2. Abra el archivo al llamar a **open()** en la instancia de **ifstream** creada en el paso 1. Como opción, puede abrir el archivo al mismo tiempo que crea el objeto de **ifstream**. (Consulte la sección *Análisis* de esta solución.)

3. Confirme que el archivo se ha abierto correctamente.
4. Una manera de leer datos no formados en el archivo consiste en llamar a **read()**.
5. Cierre el archivo al llamar a **close()**.
6. Confirme que las operaciones de escritura han sido correctas. Esto puede hacerse al llamar a **good()** en el flujo de entrada.

Análisis

Una revisión general de la apertura y el cierre de un archivo se encuentra en *Apertura y cierre de un archivo*, casi al principio de este capítulo. Aquí se presentan los detalles específicos relacionados con el uso de **ifstream** para leer datos binarios, sin formato.

Para realizar entrada de datos binarios sin formato, debe tener un objeto de tipo **ifstream** que dé soporte a operaciones binarias. La clase **ifstream** usa el encabezado **<fstream>** y define estos dos constructores:

ifstream()

explicit ifstream(const char **nombrear*, ios::openmode *modo* = ios:in)

El primero crea una instancia de **ifstream** que no está vinculada aún con un archivo. El segundo crea una instancia de **ifstream** y luego abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que *modo* tiene como opción predeterminada **ios::in**, pero no incluye la marca **ios::binary**. Como opción predeterminada, los archivos se abren en modo de texto. Para abrirlo para entrada binaria, el argumento *modo* debe especificar **ios::in** y **ios::binary**. Por ejemplo, lo siguiente abre un archivo llamado **prueba.dat** para entrada binaria:

```
ifstream archentrada("prueba.dat", ios::in | ios::binary);
```

Cuando se especifica la marca de modo **binary**, los datos se leen byte por byte en su forma binaria, sin trabajar. Esto evita posibles traducciones de caracteres (como la conversión de nueva línea en la secuencia retorno de carro/avance de línea) que podría ocurrir cuando el archivo se abre en modo de texto. (Si no se especifica **ios::binary**, el archivo se abre de manera automática en modo de texto.) Si no se usa el modo binario, puede llevar a que la información leída sea diferente de la que se encuentra en el archivo. Por tanto, siempre debe especificar **ios::binary** cuando abra un archivo para entrada binaria.

Si utiliza el constructor predeterminado, entonces necesitará vincular un archivo con la instancia de **ifstream** después de construirla. Para esto, llame a **open()**. Aquí se muestra la versión definida por **ifstream:**

void open(const char **nombrear*, ios::openmode *modo* = ios::in)

Abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que, al igual que el constructor **ifstream**, la opción predeterminada de *modo* es **ios::in**. Por tanto, debe especificar explícitamente **ios::in** y **ios::binary** para leer datos binarios sin formato.

Antes de tratar de leer el archivo, debe confirmar que se ha abierto. Puede hacer esto de diversas maneras. El método usado en esta solución consiste en aplicar el operador **!** a la instancia de **ifstream**. Recuerde que el operador **!** devuelve la salida de una llamada a **fail()** en el flujo. Por tanto, si devuelve true, la operación de apertura ha fallado.

Una manera de leer salida sin formato de un archivo consiste en usar la función **read()**. Lee un bloque de datos en un flujo. Aquí se muestra:

istream &read(const char **buf*, streamsize *num*)

Aquí, *buf* es un apuntador al bloque de memoria (como una matriz) en que se almacenará la entrada. El número de bytes que se leerá se especifica con *num*. El tipo **streamsize** está definido como alguna forma de entero que puede contener el número más grande de bytes que es posible transferir en cualquier operación de E/S. La función devuelve una referencia al flujo. Si es menor que el número especificado de bytes disponible (lo que sucederá si trata de leer al final del archivo), **read()** leerá menos de *num* bytes y **failbyte** se enviará en el flujo que invoca (lo que indica un error). Aunque *buf* está especificado como **char** *, puede usar **read()** para leer cualquier tipo de datos binarios. Simplemente convierta un apuntador a los datos a **char** * y especifique la longitud del bloque en bytes. (Recuerde que, en C++, un **char** es siempre exactamente de un byte de largo.) En el programa de ejemplo se muestra este proceso.

Cuando haya terminado de leer en un archivo, debe cerrarlo. Esto se hace al llamar a **close(),** que se muestra aquí:

void close()

El archivo se cierra automáticamente cuando se llama al destructor de **ifstream**. Sin embargo, a manera de ejemplo, en este libro se llamará explícitamente a **close()** en todos los casos.

En esta solución se comprueba que no han ocurrido errores de E/S al llamar a **good()** en el flujo. Aquí se muestra:

bool good() const

Devuelve true si no están establecidas marcas de error.

Ejemplo

En el siguiente ejemplo se demuestra cómo leer datos binarios sin formato. Se hace al leer el archivo **InvDat.dat** creado por el programa de ejemplo en *Escriba datos binarios sin formato en un archivo.* Este archivo contiene tres estructuras de **inventario**. Después de la llamada a **read()**, la matriz **inv** contendrá el mismo patrón de bytes que el almacenado en el archivo.

```
// Usa read() para dar entrada a bloques de datos binarios.
//
//   Este programa lee el archivo InvDat.dat
//   que se creó en el programa de ejemplo de
//
//   Escriba datos binarios sin formato en un archivo

#include <iostream>
#include <fstream>

using namespace std;

// Una estructura simple de inventario.
struct inventario {
  char producto[20];
```

```
  int cantidad;
  double costo;
};

int main()
{
  // Abre el archivo para entrada binaria.
  ifstream archentrada("InvDat.dat", ios::in | ios::binary);

  // Confirma que el archivo se abrió sin error.
  if(!archentrada) {
    cout << "No se puede abrir el archivo.\n";
    return 1;
  }

  inventario inv[3];

  // Lee bloques de datos binarios.
  for(int i=0; i<3; i++)
    archentrada.read((char *) &inv[i], sizeof(inventario));

  // Cierra el archivo.
  archentrada.close();

  // Confirma que no hubo errores de archivo.
  if(!archentrada.good()) {
    cout << "Ha ocurrido un error con el archivo.\n";
    return 1;
  }

  // Despliega los datos de inventario leídos del archivo.
  for(int i=0; i < 3; i++) {
    cout << inv[i].producto << "\n";
    cout << " cantidad en existencia: " << inv[i].cantidad;
    cout << "\n costo: " << inv[i].costo << "\n\n";
  }

  return 0;
}
```

Aquí se muestra la salida:

```
Martillos
 cantidad en existencia: 3
 costo: 99.95

Pinzas
 cantidad en existencia: 12
 costo: 78.55

Llaves
 cantidad en existencia: 19
 costo: 27.55
```

Opciones

Como ya se explicó, **read()** lee un número especificado de bytes de un archivo. Sin embargo, si solicita más bytes de los disponibles en el archivo (como cuando lee cerca del final o en el final del archivo), **read()** obtendrá menos bytes del número solicitado. Puede determinar cuántos bytes realmente se leyeron al llamar a **gcount()**. Se muestra a continuación:

streamsize gcount() const

Devuelve el número de caracteres leído por una llamada anterior a **read()**, o a cualquier otra función de entrada sin formato. Puede ver la función **gcount()** en acción en el *Ejemplo adicional* de la solución *Detección de EOF*.

En el ejemplo anterior se usó **good()** para revisar errores, pero hay varias opciones. Consulte *Revisión de errores* en la revisión general y la solución *Use excepciones para detectar y manejar errores de E/S* para conocer más detalles. También puede revisar errores de entrada para monitorear el estado del flujo. Debido a que **read()** devuelve una referencia al flujo sobre el que está operando, puede aplicar el operador ! al objeto devuelto. Recuerde que cuando se aplica ! a un flujo, devuelve el resultado de **fail()** aplicado al mismo flujo. Por tanto, puede probar el éxito de una llamada a **read()** de la manera siguiente:

```
if(!read(...)) { // ... maneja el error de lectura
```

Por ejemplo, en el programa anterior, puede usar la siguiente secuencia para leer los registros del inventario, confirmando el éxito de cada operación de lectura en el proceso:

```
// Lee bloques de datos binarios.
for(int i=0; i<3; i++)
  if(!archentrada.read((char *) &inv(i), sizeof(inventario))) {
    cout <<< "Error al leer el archivo.";
    // maneja el error ...
  }
```

El hecho de tomar este método para revisar errores afina su código fuente. Sin embargo, debido a que cada llamada a **read()** también da como resultado que se evalúe una instrucción **if** (lo que toma tiempo), *no* afina el rendimiento de su programa. Como regla general, las excepciones ofrecen una mejor opción en este tipo de situación.

Otra manera de leer entrada sin formato, basada en caracteres, consiste en usar la función **get()** o **getline()**. Se describen en *Use **get()** y **getline()** para leer de un archivo.*

En algunas situaciones de entrada, querrá leer datos hasta que llegue al final del archivo. Puede determinar cuando se ha encontrado el final de un archivo al llamar a **eof()** en el flujo. Consulte *Detección de EOF*.

Para leer datos formados, abra el flujo de entrada en modo de texto. Consulte *Lea datos formados de un archivo de texto* para conocer más detalles. Para escribir datos formados en un archivo, use **ofstream**. Consulte *Escriba datos formados en un archivo de texto*. Para abrir un archivo para entrada y salida, cree un objeto de **fstream**. Consulte *Lea un archivo y escriba en él.*

Use get() y getline() para leer un archivo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ifstream>	ifstream	istream &get(char &*car*) istream &get(char **buf*, streamsize *num*) istream &getline(char **buf*, streamsize *num*)

En la solución anterior se describió la manera de leer datos binarios sin formato mediante el uso de la función **read()**. Esta función es especialmente útil cuando se leen bloques de datos, como en el programa de ejemplo de la solución anterior. Sin embargo, cuando se leen datos **char**, como caracteres individuales o líneas de texto, las funciones **get()** y **getline()** pueden resultar más convenientes. En esta solución se muestra cómo usarlas.

Paso a paso

Para leer caracteres de un archivo usando **get()** se requieren estos pasos:

1. Abra el archivo para entrada. Puede abrirse en modo de texto o binario. Sin embargo, esté consciente de que si el archivo se abre en modo de texto, puede presentarse cierta traducción de caracteres, como la conversión de caracteres de nueva línea en secuencias retorno de carro/avance de línea.
2. Una manera de leer un solo carácter consiste en usar **get(char &*car*)**.
3. Una manera de leer una secuencia de caracteres consiste en usar **get(char **buf*, streamsize *num*)**.
4. Confirme que las operaciones de lectura han tenido éxito.

Para leer una línea completa de texto con el uso de **getline()** se requieren estos pasos:

1. Abra el archivo para entrada. Puede abrirse en modo de texto o binario. Sin embargo, esté consciente de que si el archivo se abre en modo de texto, puede presentarse cierta traducción de caracteres, como la conversión de caracteres de nueva línea en secuencias retorno de carro/avance de línea.
2. Una manera de leer una línea que termina en un carácter de nueva línea consiste en llamar a **getline(char **buf*, streamsize *num*)**.
3. Confirme que las operaciones de lectura han tenido éxito.

Análisis

En las soluciones anteriores se describieron los pasos necesarios para abrir un archivo para entrada o salida en modo de texto o binario. Consulte esas soluciones para conocer detalles sobre la apertura de un archivo.

Hay varias versiones de **get()**. Aquí se muestran las dos usadas en esta solución:

istream &get(char &*car*)

istream &get(char **buf*, streamsize *num*)

La primera forma lee un sólo carácter del flujo que invoca, y coloca ese valor en *car*. La segunda forma lee caracteres de una matriz señalada por *buf* hasta que se han leído *num*-1 caracteres, se ha encontrado un carácter de nueva línea o se ha llegado al final del archivo. La matriz a la que señala *buf* estará terminada por un carácter nulo por **get()**. Si se encuentra el carácter de nueva línea en el flujo de entrada, *no* se extrae. En cambio, permanece en el flujo hasta la siguiente operación de entrada. Ambas devuelven una referencia al flujo.

La función **getline()** tiene las dos formas. La usada en esta solución se muestra aquí:

istream &getline(char **buf*, streamsize *num*)

Lee caracteres en la matriz señalada por *buf* hasta que se han leído los caracteres *num*-1, se ha encontrado una nueva línea de caracteres, o se ha llegado al final del archivo. La matriz a la que señala *buf* estará terminada por un carácter nulo por **getline()**. Si se encuentra el carácter de nueva línea en el flujo de entrada, se extrae pero no se coloca en *buf*. La función devuelve una referencia al flujo.

Como se observa, **getline()** es casi idéntica a la versión **get(*buf*, *num*)** de **get()**. Ambas leen caracteres de la entrada y los colocan en la matriz a la que señala *buf* hasta que se han leído *num*-1 caracteres o se ha encontrado un carácter de nueva línea. La diferencia es que **getline()** lee y elimina el carácter de nueva línea del flujo de entrada; **get()** no lo hace.

Es importante comprender que **get()** y **getline()** pueden usarse en archivos abiertos en modo de texto o binario. La única diferencia es que si el archivo está abierto en modo de texto, puede ocurrir cierta traducción de caracteres, como la conversión de nuevas líneas en secuencias retorno de carro/avance de línea.

Cuando se usa **get()** o **getline()**, debe asegurarse de que la matriz que estará recibiendo entrada sea lo suficientemente grande como para contener la entrada que recibirá. Por tanto, debe ser por lo menos del mismo largo que la cuenta de caracteres pasada en *num*. Si se pasa por alto esta regla, puede producirse un desbordamiento de búfer, que probablemente hará que el programa deje de funcionar. También representa una posible amenaza a la seguridad porque deja su aplicación abierta al famoso "ataque de desbordamiento de búfer". En general, debe ejercerse cuidado extremo cuando se incluyen datos en una matriz.

Tiene la opción de confirmar el éxito de **get()** o **getline()** de la misma manera que lo haría al llamar a **read()**. Consulte la solución anterior y *Revisión de errores*, en la revisión general casi al principio de este capítulo, para conocer más detalles.

Ejemplo

En el siguiente ejemplo se muestran **get()** y **getline()** en acción.

```
// Usa get() y getline() para leer caracteres.

#include <iostream>
#include <fstream>

using namespace std;

int main()
{
  char car;
  char cad[256];

  // Primero, escribe algunos datos en un archivo.
```

```
ofstream archsalida("prueba.dat");
if(!archsalida) {
  cout << "No se puede abrir el archivo para salida.\n";
  return 1;
}

// Escribe en el archivo.
archsalida << "Veamos una l\u00a1nea de texto.\n";
archsalida << "Y ahora otra l\u00a1nea de texto.\n";
archsalida << "Y al final la \u00a3ltima l\u00a1nea de texto.\n";

archsalida.close();
if(!archsalida.good()) {
  cout << "Ha ocurrido un error mientras se escrib\u00a1a en el archivo .\n";
  return 1;
}

// Ahora, abre el archivo para entrada.
ifstream archentrada("prueba.dat", ios::in);
if(!archentrada) {
  cout << "No se puede abrir el archivo para entrada.\n";
  return 1;
}

cout << "Usa get():\n";

// Obtiene los tres primeros caracteres del archivo.
cout << "S\u00a2lo son los tres primeros caracteres: ";
for(int i=0; i < 3; ++i) {
  archentrada.get(car);
  cout << car;
}
cout << endl;

// Ahora, usa get() para leer el final de la línea.
archentrada.get(cad, 255);
cout << "Esto es el resto de la primera l\u00a1nea: ";
cout << cad << endl;

// Debido a que la llamada anterior a get() no eliminó
// el carácter de nueva línea del flujo de entrada, debe
// eliminarse con otra llamada a get(car):
archentrada.get(car);

cout << "\nAhora se usa getline():\n";

// Por último, usa getline() para leer las dos líneas siguientes de texto.
archentrada.getline(cad, 255);
cout << cad << endl;
archentrada.getline(cad, 255);
cout  << cad;

archentrada.close();
if(!archentrada.good()) {
  cout << "Ha ocurrido un error mientras se lee o se cierra el archivo.\n";
```

```
    return 1;
  }

  return 0;
}
```

Aquí se muestra la salida:

```
Usa get():
Sólo son los tres primeros caracteres: Vea
Esto es el resto de la primera línea: mos una línea de texto.

Ahora se usa getline():
Y ahora otra línea de texto.
Y al final la última línea de texto.
```

En el programa, observe esta secuencia:

```
// Ahora, usa get() para leer el final de la línea.
archentrada.get(cad, 255);
cout << "Esto es el resto de la primera l\u00a1nea: ";
cout << cad << endl;

// Debido a que la llamada anterior a get() no eliminó
// el carácter de nueva línea del flujo de entrada, debe
// eliminarse con otra llamada a get(car):
archentrada.get(car);
```

Como se explicó, la versión **get(*buf*, *num*)** de **get()** no elimina un carácter de nueva línea del flujo de entrada. Por tanto, la nueva línea se leerá en la siguiente operación de entrada. A menudo, como pasa con el programa de ejemplo, es necesario eliminar y descartar el carácter de nueva línea. Esto se maneja con la llamada a la versión de **get(*car*)***.

Opciones

Hay otra forma de **get()** que proporciona una opción cuando sólo se lee un carácter. Se muestra a continuación:

int get()

Esta forma de **get()** devuelve el siguiente carácter del flujo. Devuelve un valor que representa el final del archivo si se ha alcanzado éste. Para flujos basados en **char**, como **ifstream**, el valor EOF es **char_traits<char>::eof()**.

Cuando se lee una secuencia de caracteres mediante **get()**, puede especificar el delimitador al emplear esta forma:

istream &get(char **buf*, streamsize *num*, char *delim*)

Funciona igual que **get(*buf*, *num*)** descrito en esta solución, excepto que detiene la lectura cuando se encuentra el carácter pasado en *delim* (o cuando se han leído *num*-1 caracteres o se ha alcanzado el final del archivo).

***Nota del revisor técnico:** Observe que al escribir las secuencias de escape, se escriben de manera diferente en el archivo y su presentación en pantalla. Por razones de consistencia, se prefiere que la salida a la pantalla sea la correcta. Se sugiere explorar las opciones de configuración regional y de idioma de C++ para tratar adecuadamente este tema.

Cuando se lee una línea de texto mediante **getline()**, puede especificar el delimitador mediante el uso de esta forma:

istream &getline(char **buf*, streamsize *num*, char *delim*).

Funciona igual que **getline(*buf*, *num*)** descrito en esta solución, excepto que detiene la lectura cuando se encuentra el carácter pasado en *delim* (o cuando se han leído *num*-1 caracteres o se ha alcanzado el final del archivo).

Lea un archivo y escriba en él

Componentes clave		
Encabezados	**Clases**	**Funciones**
<fstream>	fstream	void close() ostream &flush() istream &get(char &*car*) bool good() const void open(const char **nombrear*, ios::openmode modo = ios::in | ios::out) ostream &put(char *car*)

Es posible abrir un archivo para que pueda usarse con entrada y salida. Esto suele hacerse cuando un archivo de datos necesita actualizarse. En lugar de volver a escribir todo el archivo, puede escribir sólo una pequeña parte de él. Esto resulta especialmente valioso en archivos que usan registros de longitud fija, porque ofrece una manera conveniente de actualizar un registro sin reescribir todo el archivo. Por supuesto, abrir un archivo para entrada y salida resulta útil en otras situaciones, como cuando quiere leer el contenido de un archivo, modificarlo y luego volver a escribir el contenido modificado en el mismo archivo. Al usar un archivo abierto para entrada y salida, sólo necesita abrir y cerrar el archivo una vez, con lo que se afina su código. Cualquiera que sea su propósito, en esta solución se muestra el procedimiento básico necesario para leer un archivo y escribir en él.

Paso a paso

Para realizar operaciones de entrada y salida en un archivo se requieren los siguientes pasos:

1. Abra el archivo para lectura y escritura al crear un objeto de tipo **fstream**. La clase **fstream** hereda **ifstream** y **ofstream**. Esto significa que permite operaciones de entrada y salida.
2. Use las funciones de salida definidas por **ofstream** para escribir en el archivo. La que se usa en esta solución es **put()**.
3. Use las funciones de entrada definidas por **ifstream** para leer el archivo. La que se usa en esta solución es **get()**.

4. Para muchas implementaciones de compilador, cuando cambia entre entrada y salida, necesitará llamar a **seekg()**, **seekp()** o **flush()**. En esta solución se usa **flush()**.
5. Cierre el archivo.
6. Confirme que las operaciones de entrada y salida fueron correctas. Esto puede hacerse al llamar a **good()** en el flujo de entrada o de varias otras maneras.

Análisis

Una revisión general de la apertura y el cierre de un archivo se encuentra en *Apertura y cierre de un archivo*, casi al principio de este capítulo. Aquí se presenta la información relacionada específicamente con **fstream**.

La clase **fstream** hereda la clase **iostream**, que hereda **istream** y **ostream**. Esto permite el soporte de operaciones de entrada y salida. Más aún, todas las técnicas descritas en las soluciones anteriores, como leer y escribir de un **ifstream** u **ofstream**, se aplican a **fstream**. La única diferencia es que **fstream** da soporte a lectura y escritura.

Para realizar operaciones de entrada/salida, debe tener un objeto de tipo **fstream** que dé soporte a operaciones de entrada y salida. La clase **fstream** usa el encabezado **<fstream>** y define estos dos constructores:

fstream()

explicit fstream(const char **nombrear*, ios::openmode *modo* = ios:in | ios::out)

El primero crea una instancia de **fstream** que no está vinculada aún con un archivo. El segundo crea una instancia de **fstream** y luego abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que *modo* tiene como opción predeterminada **ios::in** y **ios::out**. Además, observe que no incluye la marca **ios::binary**. Por tanto, como opción predeterminada, el archivo se abre en modo de texto. Para abrirlo para E/S binaria, incluya la marca **ios::binary**. Cuando un archivo se abre en modo de texto, pueden ocurrir traducciones de caracteres, como el reemplazo de nueva línea por la secuencia retorno de carro/avance de línea. La apertura del texto en modo binario evita estas traducciones.

Si utiliza el constructor predeterminado, entonces necesitará vincular un archivo con la instancia de **fstream** después de que se construya al llamar a **open()**. Aquí se muestra la versión definida por **fstream**:

void open(const char **nombrear*, ios::openmode *modo* = ios:in | ios::out)

Abre el archivo especificado por *nombrear* con el modo especificado por *modo*. Observe que, como el constructor **fstream**, la opción predeterminada de *modo* es **ios::in | ios::out**. Por tanto, el archivo se abre automáticamente para operaciones predeterminadas de entrada y salida cuando *modo* está en su opción predeterminada.

Antes de tratar de escribir en el archivo, debe confirmar que el archivo está abierto. Puede hacer esto de diversas maneras. El método usado en esta solución consiste en aplicar el operador **!** a la instancia de **fstream**. Recuerde que el operador **!** devuelve la salida de una llamada a **fail()** en el flujo. Por tanto, si devuelve true, la operación de apertura ha fallado.

Una vez abierto, puede leer el archivo y escribir en él usando cualquiera de los métodos proporcionados por **istream** y **ostream**, como **get()**, **put()**, **read()** y **write()**. Estos métodos se han descrito en las soluciones anteriores.

En el caso de algunos compiladores, necesita limpiar la salida al llamar a **flush()** o realizar una operación de búsqueda al llamar a **seekg()** o **seekp()** cuando se cambia entre operaciones de

lectura y escritura. En esta solución se utiliza **flush()**. (Para conocer más detalles sobre **seekg()** y **seekp()**, consulte *Utilice E/S de archivo de acceso aleatorio*.) El método **flush()** está definido por **ostream** y se muestra a continuación:

ostream &flush()

Limpia el búfer de salida. Esto asegura que el contenido del búfer se escriba en el archivo. El sistema de E/S de C++ utiliza búferes para mejorar la eficiencia de las operaciones con archivos. Para la entrada, los datos se leen del archivo, de búfer en búfer. Cuando se alcanza el final del búfer de entrada, se lee la información del siguiente búfer. En el caso de la salida, cuando escribe sus datos, en realidad se escriben en un búfer de salida. Sólo cuando el búfer está lleno los datos se escriben físicamente en un archivo. La función **flush()** modifica este comportamiento y hace que el contenido actual del búfer se escriba en el archivo, esté lleno el búfer o no. Esto asegura que el contenido del archivo refleje inmediatamente cualquier operación de escritura que haya tenido lugar. En lo que se relaciona con los archivos de lectura/escritura, la llamada a **flush()** después de que ha escrito el archivo asegura que las operaciones de lectura reflejen el estado real del archivo:

Ejemplo

En los siguientes ejemplos se muestra cómo abrir un archivo de texto llamado **prueba.dat** para lectura y escritura. Es necesario que el archivo **prueba.dat** exista. Después de que abre el archivo, escribe tres "X" al principio del archivo. A continuación limpia el búfer de salida y luego lee los siguientes diez caracteres del archivo.

```
// Usa fstream para leer un archivo y escribir en él.

#include <iostream>
#include <fstream>

using namespace std;

int main()
{
  char car;

  // Abre un archivo para operaciones de entrada y salida.
  fstream archentradasalida("prueba.dat");

  if(!archentradasalida) {
    cout << "No se puede abrir el archivo para salida.\n";
    return 1;
  }

  // Escribe tres X.
  for(int i=0; i < 3; ++i) archentradasalida.put('X');

  if(!archentradasalida.good()) {
    cout << "Ha ocurrido un error mientras se escrib\u00a1a en el archivo.\n";
    return 1;
  }

  // Limpia el búfer de salida.
```

```
  archentradasalida.flush();

  // Obtiene los siguientes 10 caracteres del archivo.
  cout << "Aqu\u00a1 se muestran los diez caracteres siguientes: ";
  for(int i=0; i < 10; ++i) {
    archentradasalida.get(car);
    cout << car;
  }
  cout << endl;

  if(!archentradasalida.good()) {
    cout << "Ha ocurrido un error mientras se le\u00a1a el archivo.\n";
    return 1;
  }

  archentradasalida.close();

  if(!archentradasalida.good()) {
    cout << "Ha ocurrido un error mientras se cerraba el archivo.\n";
    return 1;
  }

  return 0;
}
```

Suponiendo que **prueba.dat** contiene lo siguiente:

```
abcdefghijklmnop
```

el programa producirá esta salida:

```
Aquí se muestran los diez caracteres siguientes: defghijklm
```

y el contenido de **prueba.dat** cambiará a:

```
XXXdefghijklmnop
```

Opciones

Para realizar operaciones de entrada/salida en un archivo, no hay en realidad ninguna opción adicional al uso de **fstream**.

Detección de EOF

Componentes clave		
Encabezados	**Clases**	**Funciones**
<fstream>	ifstream	bool eof() const

En algunos casos, querrá saber cuándo se ha alcanzado el final del archivo. Por ejemplo, si está leyendo una lista de valores de un archivo, entonces tal vez quiera seguir leyendo hasta que ya no haya más valores. Para esto debe contar con alguna manera de saber cuándo se ha alcanzado el final del archivo. Por fortuna, el sistema de E/S de C++ proporciona una función para hacer esto: **eof()**. En esta solución se muestra cómo usarla.

Paso a paso

Para detectar EOF se requieren estos pasos:

1. Abra el archivo que se leerá para entrada.
2. Empiece a leer datos del archivo.
3. Después de cada operación de entrada, determine si se ha alcanzado el final del archivo al llamar a **eof()**.

Análisis

La función **eof()** determina si se ha alcanzado el final del archivo. Está declarada por **istream**, que se hereda de **ifstream**. Se muestra a continuación:

bool eof() const

Devuelve true si se ha encontrado el final del flujo; de lo contrario, devuelve false.

Hay un aspecto importante del sistema de E/S de C++ que se relaciona con el final del archivo. Cuando se hace un intento por leer al final del archivo, se establecen **ios::eofbit** e **ios::failbit**. Por tanto, el encuentro del final del archivo también se considera una condición de error, aunque eso sea lo que se pretende. Más aún, si quiere detectar una falla de entrada causada por algo diferente del encuentro del final del archivo, entonces necesitará probar explícitamente esto al excluir que se revise la condición de final de archivo. Por ejemplo, esta instrucción if se presenta si está establecida **badbit** o **failbit**, pero no **eofbit**:

```
if(!archentrada.eof() && (archentrada.fail() || archentrada.bad())) { // ...
```

Recuerde que una operación de entrada puede fallar por muchas razones. El encuentro del final del archivo es sólo una de ellas.

Debido a que las marcas de estado de E/S permanecen hasta que se limpian, el encuentro del final del archivo causará que **good()** devuelva false, aunque usted haya causado a propósito esa condición. Necesita tomar esto en cuenta cuando busque y maneje errores. Por ejemplo, después de que se ha encontrado el final del archivo, puede usar la función **clear()** para restablecer las marcas de E/S. Consulte *Revisión de errores* en la revisión general, para conocer más detalles, incluidas las funciones **clear()**, **good()**, **bad()** y **fail()**.

Ejemplo

En el siguiente ejemplo se demuestra **eof()**. Crea un programa que lee y despliega el contenido de un archivo de texto. Utiliza **eof()** para saber cuando se ha leído todo el archivo. Observe que utiliza la función **get()** definida por **istream**. Se describe en *Use **get()** y **getline()** para leer un archivo*.

```
// Usa eof() para leer y desplegar un archivo de texto.
//
// El nombre del archivo se especifica en la línea de
```

```
// comandos. Por ejemplo, suponiendo que este programa
// se llama Mostrar, la siguiente línea de comandos
// desplegará el archivo llamado prueba.txt:
//
//    Mostrar prueba.txt
//

#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char *argv[])
{
  char car;

  if(argc != 2) {
    cout << "Uso: Mostrar <nombrearchivo>\n";
    return 1;
  }

  // Crea un objeto de ifstream y trata de abrir el archivo.
  ifstream archentrada(argv[1]);

  // Verifica que el archivo se abrió correctamente.
  if(!archentrada) {
    cout << "No se puede abrir el archivo.\n";
    return 1;
  }

  do {
    // Lee el siguiente carácter, si lo hay.
    archentrada.get(car);

    // Revisa si hay errores NO causados por alcanzar EOF.
    if(!archentrada.eof() && (archentrada.fail() || archentrada.bad())) {
      cout << "Error en la entrada\n";
      archentrada.close();
      return 1;
    }

    // Si aún no se encuentra EOF, despliega el siguiente carácter.
    if(!archentrada.eof()) cout << car;
  } while(!archentrada.eof());

  // Limpia los bits eof y fail.
  archentrada.clear();

  // Cierra el archivo de entrada.
  archentrada.close();

  // Confirma que el archivo se cerró sin error.
  if(!archentrada.good()) {
    cout << "Error al cerrar el archivo.";
```

```
    return 1;
  }

  return 0;
}
```

Observe que el programa revisa errores de entrada que no están relacionados con una condición de final de archivo. Esto permite que el programa informe si sucedió algo inesperado cuando leyó el archivo. Después de que se encuentra el final del archivo, los bits de estado de E/S se limpian y se cierra el archivo. Esto nos permite confirmar que la operación de cierre se dio sin error. Por supuesto, sus propias aplicaciones determinarán cómo revisar los errores. En el programa siguiente se muestra sólo un ejemplo.

Ejemplo adicional: una utilería simple de comparación de archivos

En el siguiente programa se le da un buen uso a **eof()**. Crea una utilería simple que compara dos archivos. Abre ambos para entrada binaria. Esto significa que el programa puede usarse en archivos de texto y binarios, como ejecutables. Compara los dos archivos al leer un búfer de datos de cada uno mediante el uso de **read()** y luego compara el contenido de los búferes. Utiliza **eof()** para determinar cuando ambos archivos se han leído por completo. Si los archivos tienen diferentes longitudes, o si su contenido no coincide, los archivos difieren. De otra manera, son iguales. Observe que el programa usa la función **gcount()** para determinar cuántos bytes de datos se han obtenido con **read()**. Cundo se realiza entrada al final del archivo, el número de bytes leídos puede ser menos del solicitado en la llamada a **read()**.

```
// Una utilería simple de comparación de archivos.

#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[])
{
  bool igual = true;
  bool errarch = false;

  unsigned char buf1[1024], buf2[1024];

  if(argc!=3) {
    cout << "Uso: comparchivos <archivo1> <archivo2>\n";
    return 1;
  }

  // Abre ambos archivos para operaciones binarias.
  ifstream arch1(argv[1], ios::in | ios::binary);
  if(!arch1) {
    cout << "No se puede abrir " << argv[1] << endl;
    return 1;
  }

  ifstream arch2(argv[2], ios::in | ios::binary);
  if(!arch2) {
```

```
    cout << "No se puede abrir " << argv[2] << endl;
    arch1.close();
    if(!arch1.good())
      cout << "Error al cerrar " << argv[1] << endl;
    return 1;
  }

  cout << "Comparando archivos...\n";

  do {

    // Lee un búfer completo de datos de cada archivo.
    arch1.read((char *) buf1, sizeof buf1);
    arch2.read((char *) buf2, sizeof buf2);

    // Revisa errores de lectura.
    if(!arch1.eof() && !arch1.good()) {
      cout << "Error al leer " << argv[1] << endl;
      errarch = true;
      break;
    }
    if(!arch2.eof() && !arch2.good()) {
      cout << "Error al leer " << argv[2] << endl;
      errarch = true;
      break;
    }

    // Si la longitud de los dos archivos es diferente, entonces
    // al final del archivo, las gcount serán diferentes.
    if(arch1.gcount() != arch2.gcount()) {
      cout << "Los archivos tienen diferente longitud.\n";
      igual = false;
      break;
    }

    // Compara el contenido de los búferes.
    for(int i=0; i < arch1.gcount(); ++i)
      if(buf1[i] != buf2[i]) {
        cout << "Los archivos son diferentes.\n";
        igual = false;
        break;
      }

  } while(!arch1.eof() && !arch2.eof() && igual);

  if(!errarch && igual) cout << "Los archivos son iguales.\n";

  // Limpia eofbit, y tal vez bits de error.
  arch1.clear();
  arch2.clear();

  arch1.close();
  arch2.close();
```

```
  if(!arch1.good() || !arch2.good()) {
    cout << "Error al cerrar los archivos.\n";
    return 1;
  }

  return 0;
}
```

Opciones

Puede detectar el final de archivo de varias maneras. En primer lugar, puede usar la función **rdstate()**, que devuelve todas las marcas de estado en la forma de una máscara de bits. Luego puede probar el final de archivo al vincular con operaciones lógicas OR mediante **ios::eofbit** con el valor devuelto por **rdstate()**. (Esta función se describe en *Revisión de errores*.)

Si utiliza esta forma de **get()**

int get()

entonces el valor obtenido de **ifstream::traits_type::eof()** se devuelve cuando se encuentra el final del archivo. El **typedef traits_type** especifica valores asociados con el tipo de carácter usado por el flujo, que son **char** en el caso de **ifstream**. Por tanto, cuando se usa esta forma de **get()**, la siguiente secuencia detecta el final del archivo:

```
car = archentrada.get();
if(car == ifstream::traits_type::eof()) cout << "EOF encontrado";
```

¡Por supuesto, es mucho más fácil usar la función **eof()** definida por **ifstream**!

Use excepciones para detectar y manejar errores de E/S

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>	ios	void exceptions(iostate exc)
<ios>	ios_base::failure	const char *what() const

El sistema de E/S de C++ le da dos maneras de revisar errores. En primer lugar, puede usar las funciones **good()**, **bad()**, **fail()** y **rdstate()** para interrogar explícitamente las marcas de estado. Este método se describe en *Revisión de errores*, en la revisión general que se hizo al principio del capítulo. También es el método usado en casi todas las soluciones de este capítulo, porque es la manera en que se detectan los errores, como opción predeterminada. La segunda manera incluye el uso de excepciones. En este método, un error de E/S causa que se lance una excepción. Su código puede capturar esta excepción y tomar la acción apropiada para manejar el error. En esta solución se muestra cómo usar las excepciones para detectar y manejar los errores de E/S.

Paso a paso

Para detectar y manejar errores de E/S mediante el uso de excepciones se requieren los siguientes pasos:

1. En el flujo que desee monitorear, en busca de errores, llame a la función **exceptions()**, pasándola en una máscara de bits **iostate** que contiene la marca o las marcas de la excepción o las excepciones que desee para generar errores.
2. Realice operaciones de E/S desde el interior de un bloque **try**.
3. La instrucción **catch** del bloque **try** debe capturar excepciones de tipo **failure**. Es el tipo de excepción generada por el sistema de E/S.
4. Para determinar qué tipo de falla ocurrió, llame a **what()** en el objeto de excepción.

Análisis

Como opción predeterminada, el sistema de E/S no lanza una excepción cuando ocurre un error. Por tanto, para usar excepciones, debe solicitar explícitamente su uso. Más aún, debe especificar cuáles tipos de errores lanzarán una excepción. Para ello, utilizará la función **exceptions()**. Está definida por **ios_base** y es heredada por todas las clases de flujo. Aquí se muestra:

void exceptions(iostate *exc*)

Aquí, *exc* es una máscara de bits que contiene valores **iostate** que representan la condición que lanzará una excepción. Estos valores son **ios_base::failbit**, **ios_base::badbit**, **ios_base::goodbit** y **ios_base::eofbit**. Como se relacionan con flujos **char**, suele hacerse referencia a estos valores como **ios::failbit**, **ios::badbit**, **ios::goodbit** y **ios::eofbit**. Por tanto, para causar que un flujo de **char** llamado **miflujo** genere excepciones cada vez que un error cause que se establezca **failbit**, puede usar lo siguiente:

```
miflujo.exceptions(ios::failbit);
```

Después de esta llamada, cada vez que un error de E/S cause que se establezca **failbit**, se genera una excepción. Un tema adicional: como se explicó en *Revisión de errores* en la revisión general presentada casi al principio de este capítulo, el final de archivo no siempre se considera un error, en sentido estricto, pero puede usar excepciones para vigilarlo.

Una vez que haya habilitado las excepciones, debe realizar operaciones de E/S dentro de un bloque **try** que capture excepciones que tienen un tipo de base **ios_base::failure**. Observe que esta clase es una clase miembro de **ios_base**. Se declara de la manera en que se muestra a continuación:

```
class ios_base::failure : public exception |
public:
  explicit failure(const string &cad);
  virtual ~failure();
  virtual const char *what() const throw();
);
```

Observe que hereda **exception()**, que es una clase de base para todas las excepciones. La función **what()** devuelve una cadena que describe la excepción. En teoría, podría usar la cadena devuelta por **what()** para determinar lo que ocurrió. En la práctica, suele ser mejor depender de la lógica de su propio programa para realizar esta función, porque la cadena devuelta por **what()** tal vez no sea específica de la causa real del error. Por ejemplo, sólo podría establecer cuál bit de error se estableció. Más aún, esta cadena podría variar entre compiladores (y tal vez así será), o incluso entre versiones diferentes del mismo compilador. Por esto es por lo que a veces no resulta particularmente útil.

Ejemplo

En el siguiente ejemplo se muestra cómo usar excepciones para manejar errores cuando se realiza E/S. Se vuelve a trabajar el programa de ejemplo de *Escriba datos binarios sin formato en un archivo* de modo que utilice excepciones para detectar y manejar errores de E/S. Observe que cada operación de E/S (abrir el archivo, leer datos y cerrar el archivo) se realiza dentro de su propio bloque **try**. Esto facilita el responder a cada excepción en forma individualizada. Por supuesto, el método que use debe ser adecuado para su aplicación y sus necesidades específicas. Observe que el programa usa la cadena devuelta por **what()** para desplegar el error. Esto se incluye simplemente para la demostración. Excepto por la depuración, normalmente no desplegaría esta cadena.

```
// Usa excepciones para vigilar y manejar errores de E/S.
//
//  En este programa se vuelve a trabajar el programa de:
//
//    Escriba datos binarios sin formato en un archivo
//
// De modo que utilice excepciones para detectar y manejar errores de E/S.

#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

// Una estructura simple de inventario.
struct inventario {
  char producto[20];
  int cantidad;
  double costo;
};

int main()
{
  int completion_status = 0;

  // Crea un flujo de salida.
  ofstream archsalida;

  // Habilita la excepción para errores de E/S.
  archsalida.exceptions(ios::failbit | ios::badbit);

  // Trata de abrir el archivo para salida binaria.
```

```
  try {
    archsalida.open("InvDat.dat", ios::out | ios::binary);
  } catch(ios_base::failure exc) {
    cout << "No se puede abrir el archivo.\n";
    cout << "La cadena devuelta por what(): " << exc.what() << endl;
    return 1;
  }

  // Crea algunos datos de inventario.
  inventario inv[3];

  strcpy(inv[0].producto,"Martillos");
  inv[0].cantidad = 3;
  inv[0].costo = 99.95;

  strcpy(inv[1].producto, "Pinzas");
  inv[1].cantidad = 12;
  inv[1].costo = 78.55;

  strcpy(inv[2].producto, "Llaves");
  inv[2].cantidad = 19;
  inv[2].costo = 27.55;

  // Escribe datos de inventario en el archivo. Si ocurre un error,
  // la excepción se manejará con la instrucción catch.
  try {
    for(int i=0; i<3; i++)
      archsalida.write((const char *) &inv[i], sizeof(inventario));
  } catch(ios_base::failure exc) {
    cout << "Ha ocurrido un error cuando se trataba de escribir en el archivo.\n";
    cout << "La cadena devuelta por what(): " << exc.what() << endl;
    completion_status = 1;
  }

  // También maneja un error que podría ocurrir cuando cierra el archivo.
  try {
    // Cierra el archivo.
    archsalida.close();
  } catch(ios_base::failure exc) {
    cout << "Ha ocurrido un error cuando se trataba de cerrar el archivo.\n";
    cout << "La cadena devuelta por what(): " << exc.what() << endl;
    completion_status = 1;
  }

  return completion_status;
}
```

He aquí algunos temas que deben quedar claros en relación con el ejemplo anterior. En primer lugar, observe que si el archivo no puede abrirse, entonces el programa se cierra. Esto es apropiado, porque si el archivo no puede abrirse, entonces no puede escribirse en él y no hay razón para seguir adelante. Más aún, debido a que el archivo no está abierto, no es necesario que se cierre. Por tanto, es apropiado salir del programa en este momento.

A continuación, observe que el manejador de excepciones para **write()** no cierra el programa. En cambio, establece la variable **completion_status** en 1 y deja que siga la ejecución del programa. En este momento, aunque haya ocurrido un error, el archivo aún está abierto y debe cerrarse. Por tanto, la ejecución sigue hasta la llamada a **close()**.

Es importante comprender que, en este ejemplo, el archivo se cerrará automáticamente cuando el programa termine, porque el destructor de **ofstream** cierra el archivo (como se explicó en la revisión general presentada en páginas anteriores de este capítulo). Sin embargo, en casi todos los programas reales, la situación no es tan fácil. Por ejemplo, si se permite al usuario volver a probar la operación de un archivo, entonces es imperativo que asegure que el intento anterior cerró el archivo. De otra manera, habrá problemas. Por ejemplo, puede volverse imposible abrir de nuevo el archivo, porque nunca se cerró. Además, el programa consume recursos del sistema, como manejadores de archivo, de los que existe un número finito. Lo importante es que, debido a que una excepción causa un cambio abrupto en el flujo normal de la ejecución, es necesario asegurar en esos casos que se cierre cualquier archivo que se haya abierto.

Opciones

Como se explicó en *Revisión de errores*, puede vigilar errores al usar las funciones **good()**, **fail()**, **rdstate()** y, en algunos casos, **eof()**. Aunque el uso de excepciones puede simplificar el manejo de errores en algunos casos, para muchos programas cortos, como los de este libro, el uso de las funciones para reporte de errores es más fácil. Esto resulta especialmente cierto cuando lo que le preocupa es que la operación general de E/S (apertura, lectura o escritura y cierre) tenga éxito. Por esto, casi todos los programas de este libro que realizan E/S de archivo usarán las funciones de reporte de errores y sin excepciones. Por supuesto, el método que use estará dictado por los aspectos y las necesidades específicas de su aplicación.

Use E/S de archivo de acceso aleatorio

Componentes clave		
Encabezados	**Clases**	**Funciones**
<fstream>	ifstream	istream &seekg(off_type *despl*, ios::seekdir, *origen*)
<fstream>	ofstream	ostream &seekp(off_type *despl*, ios::seekdir, *origen*)

En general, hay dos maneras en que puede accederse a un archivo, de manera secuencial o aleatoria. Con el acceso secuencial, el apuntador a archivo recorre el archivo de manera estrictamente lineal, de principio a fin. Con el acceso aleatorio, es posible colocar el apuntador a archivo en cualquier ubicación del archivo. Por tanto, el acceso aleatorio le permite leer de una parte específica de un archivo o escribir en ella, según se requiera o bajo pedido. Es importante comprender que cualquier archivo puede tener acceso de cualquier manera. Por tanto, el acceso aleatorio no es dependiente del archivo, sino de las funciones usadas para acceder a éste. Dicho eso, por lo general el acceso aleatorio se utilizará en un archivo que está compuesto por registros de longitud fija. Mediante el acceso aleatorio, es posible leer o escribir un registro específico. En esta solución se muestran las técnicas necesarias para usar acceso aleatorio en C++.

Paso a paso

Para usar acceso aleatorio se requieren estos pasos:

1. Abra el archivo deseado para E/S binaria.
2. Para archivos de entrada, mueva el apuntador para obtener al llamar a **seekg()**.
3. Para archivos de salida, mueva el apuntador para colocar al llamar a **seekp()**.
4. En el caso de archivo capaces de entrada y salida, use **seekg()** para mover el apuntador para obtener. Use **seekp()** para mover el apuntador para colocar.
5. Una vez que se ha establecido la ubicación, realice la operación deseada.

Análisis

El sistema de E/S de C++ administra dos apuntadores asociados con un archivo. Uno es el *apuntador para obtener*, que especifica en qué lugar del archivo ocurrirá la siguiente operación de entrada. El otro es el *apuntador para colocar*, que especifica en qué lugar del archivo ocurrirá la siguiente operación de salida. Cada vez que tiene lugar una operación de entrada o salida, el apuntador apropiado se avanza de manera secuencial y automática. Al usar las funciones de acceso aleatorio, puede colocar el apuntador para obtener o colocar a voluntad, permitiendo que el archivo se acceda de manera no secuencial.

Las funciones **seekg()** y **seekp()** cambian la ubicación de los apuntadores para colocar y obtener, respectivamente. Cada una tiene dos formas. Aquí se muestran las usadas en esta solución:

istream &seekg(off_type *despl*, ios::seekdir *origen*)

ostream &seekp(off_type *despl*, ios::seekdir *origen*)

Aquí, **off_type** es un tipo entero definido por **ios** que puede contener el valor válido más largo que *despl* puede tener. **seekdir** es una enumeración definida por **ios_base** (que es heredado por **ios**) que determina la manera en que se realizará la búsqueda.

La función **seekg()** mueve el apuntador para obtener del archivo asociado un número *despl* de caracteres a partir del *origen* especificado, que debe ser uno de tres valores:

beg	Principio del archivo
cur	Ubicación actual
end	Final del archivo

La función **seekp()** mueve el apuntador para colocar del archivo asociado un número *despl* de caracteres a partir del *origen* especificado, que debe ser uno de los valores mostrados.

La función **seekp()** se declara con **ostream** y es heredada por **ofstream**. La función **seekg()** se declara con **istream** y es heredada por **ifstream**. Tanto **istream** como **ostream** se heredan de **fstream**, que permite operaciones de entrada y salida.

Por lo general, la E/S de acceso aleatorio sólo debe realizarse en los archivos abiertos para operaciones binarias. Las traducciones de caracteres que pueden ocurrir en archivos de texto podrían causar que una solicitud de posición esté fuera de sincronía con el contenido real del archivo.

Cuando un archivo está abierto para operaciones de lectura y escritura, como cuando se usa un objeto de **fstream**, entonces por lo general debe realizar una operación de búsqueda cuando se cambia entre lectura y escritura. (Consulte *Lea un archivo y escriba en él*.)

Ejemplo

En el siguiente programa se usa **seekp()** y **seekg()** para invertir caracteres en un archivo. El nombre del archivo y el número de caracteres que se invertirá, empezando en el principio, se especifica en la línea de comandos. Debido a que son necesarias las operaciones de lectura y escritura, el archivo se abre usando **fstream**, que permite entrada y salida.

```
// Demuestra E/S de acceso aleatorio.
//
// Este programa invierte los primeros N caracteres dentro de
// un archivo. El nombre del archivo y el número de caracteres
// que se invertirá se especifica en la línea de comandos.

#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

int main(int argc, char *argv[])
{
  long n, i, j;
  char car1, car2;

  if(argc!=3) {
    cout << "Uso: Invertir <nombrearchivo> <num>\n";
    return 1;
  }

  // Abre el archivo para operaciones binarias de entrada y salida.
  fstream archentradasalida(argv[1], ios::in | ios::out | ios::binary);

  if(!archentradasalida) {
    cout << "No se puede abrir el archivo de entrada.\n";
    return 1;
  }

  // Convierte la representación de cadena del número de
  // caracteres que se invierten en un valor largo.
  n = atol(argv[2]) - 1;

  // Usa acceso aleatorio para invertir los caracteres.
  for(i=0, j=n; i < j; ++i, --j) {

    // Primero, obtiene los dos caracteres.
    archentradasalida.seekg(i, ios::beg);
    archentradasalida.get(car1);
    archentradasalida.seekg(j, ios::beg);
    archentradasalida.get(car2);
```

```
     // Now, write them to the opposite locations.
    archentradasalida.seekp(i, ios::beg);
    archentradasalida.put(car2);
    archentradasalida.seekp(j, ios::beg);
    archentradasalida.put(car1);

    // Confirma el éxito de cada ciclo de lectura y escritura.
    if(!archentradasalida.good()) {
      cout << "Error al leer o escribir caracteres.";
      archentradasalida.clear();
      break;
    }
  }

  // Cierra el archivo.
  archentradasalida.close();

  // Confirma que no ocurrieron errores cuando se cerró el archivo.
  if(!archentradasalida.good()) {
    cout << "Ha ocurrido un error con el archivo.";
    return 1;
  }

  return 0;
}
```

Para usar el programa, especifique el nombre del archivo que desee invertir, seguido por el número de caracteres que se invertirá. Por ejemplo, para invertir los primeros diez caracteres de un archivo llamado PRUEBA, utilice esta línea de comandos:

```
invertir prueba 10
```

Si el archivo hubiera contenido:

```
abcdefghijklmnopqrstuvwxyz
```

entonces el archivo contendrá lo siguiente después de que se ejecuta el programa:

```
jihgfedcbaklmnopqrstuvwxyz
```

Ejemplo adicional: use E/S de acceso aleatorio para acceder a registros de tamaño fijo

Como se mencionó, uno de los principales usos de la E/S de acceso aleatorio está en bases de datos que contienen registros de tamaño fijo. Por ejemplo, considere una base de datos que contiene información de inventario. Para encontrar una entrada específica en ese archivo, necesitará rastrearlo registro por registro. Tal vez quiera actualizar o eliminar un registro específico. Estos tipos de operaciones se facilitan mediante el uso del E/S de acceso aleatorio. El siguiente ejemplo le da una idea del proceso. Utiliza el **InvDat.dat** creado por el programa de ejemplo en *Escriba datos binarios sin formato en un archivo*. Despliega la entrada que especifique por número en la línea de comandos.

```
// Usa E/S de acceso aleatorio para leer registros específicos de
// inventario de un archivo de datos. Este programa lee el archivo
// InvDat.dat, creado por el programa de ejemplo en la solución:
//
//   Escriba datos binarios sin formato en un archivo

#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

// Una estructura simple de inventario.
struct inventario {
  char producto[20];
  int cantidad;
  double costo;
};

int main(int argc, char *argv[])
{
  inventario entrada;
  long num_registro;

  if(argc != 2) {
    cout << "Uso: MostrarRegistro <num-registro>\n";
    return 1;
  }

  // Convierte la representación de cadena del número de
  // entrada en un valor largo.
  num_registro = atol(argv[1]);

  // Confirma que el número de registro es mayor o
  // igual a cero.
  if(num_registro < 0) {
    cout << "Los n\u00a3meros de registro deben ser mayores o iguales a 0.\n";
    return 1;
  }

  // Abra el archivo para entrada binaria.
  ifstream archBDInv("InvDat.dat", ios::in | ios::binary);

  // Confirma que el archivo se abrió sin error.
  if(!archBDInv) {
    cout << "No se puede abrir el archivo.\n";
    return 1;
  }

  // Lee y despliega la entrada especificada en la línea de comandos.
  // Primero, busca el registro deseado.
  archBDInv.seekg(sizeof(inventario) * num_registro, ios::beg);

  // Luego, lee el registro.
```

```
  archBDInv.read((char *) &entrada, sizeof(inventario));

  // Cierra el archivo.
  archBDInv.close();

  // Confirma que no hubo errores de archivo.
  if(!archBDInv.good()) {
    cout << "Ha ocurrido un error con el archivo.\n";
    return 1;
  }

  // Despliega el inventario para la entrada especificada.
  cout << entrada.producto << endl;
  cout << "Cantidad en existencia: " << entrada.cantidad;
  cout << "\nCosto: " << entrada.costo << endl;

  return 0;
}
```

He aquí una ejecución de ejemplo:

```
C:>MostrarRegistro 1
Pinzas
Cantidad en existencia: 12
Costo: 78.55
```

La característica clave del programa es el uso de **seekg()** para mover al registro especificado mediante el uso de esta instrucción:

```
archBDInv.seekg(sizeof(inventario) * num_registro, ios::beg);
```

Para encontrar un registro específico, primero multiplica el tamaño de la estructura **inventario** (que es la longitud de cada registro en la base de datos) mediante el número de registros que habrá de obtenerse. Luego busca esta ubicación en el archivo. El mismo método básico puede aplicarse a cualquier archivo que contenga registros de longitud fija.

Empleando el acceso aleatorio, también es posible actualizar un registro en el lugar. Por ejemplo, en el programa anterior, si abre el archivo para entrada y salida usando el objeto **fstream**, como se muestra aquí:

```
fstream archBDInv("InvDat.dat",
                  ios_base::in | ios_base::binary | ios::out);
```

luego la siguiente secuencia cambia el registro especificado y después lee la información actualizada:

```
// Crea un nuevo artículo de inventario.
strcpy(entrada.producto, "Taladro");
entrada.cantidad = 3;
entrada.costo = 99.95;

// Establece el apuntador para colocar al inicio del registro al llamar a seekp().
archBDInv.seekp(sizeof(inventario) * num_registro, ios::beg);

// Cambia el registro.
```

```
archBDInv.write((char *) &entrada, sizeof(inventario));

// Establece el apuntador para obtener al inicio del registro al llamar a seekg().
archBDInv.seekg(sizeof(inventario) * num_registro, ios::beg);

// Luego, lee el registro actualizado.
archBDInv.read((char *) &entrada, sizeof(inventario));
```

Opciones

Puede determinar la posición actual de cada apuntador a archivo al usar estas funciones:

pos_type tellg()

pos_type tellp()

Aquí, **pos_type** es un tipo definido por **basic_ios** que puede contener el valor más grande que cualquier función puede devolver. Puede usar los valores devueltos por **tellg()** y **tellp()** como argumentos para las siguientes formas de **seekg()** y **seekp()**, respectivamente:

istream &seekg(pos_type *pos*)

ostream &seekp(pos_type *pos*)

Estas funciones le permiten guardar la posición actual del archivo, realizar otras operaciones de archivo y luego restablecer la ubicación del archivo a su posición previamente guardada.

Revise un archivo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<fstream>	ifstream	istream &ignore(streamsize *num*=1, int_type *delim* = traits_type::eof()) int_type peek() istream &unget()

Hay algunas situaciones de entrada que se facilitan al poder revisar un archivo. Por ejemplo, si un archivo contiene información contextual, entonces tal vez necesite procesar una parte de él de manera diferente a otra. C++ proporciona tres funciones que ayudan en esta tarea: **peek()**, **unget()** e **ignore()**. Le permiten obtener, pero no eliminar el siguiente carácter del archivo, devolver un carácter al flujo y omitir uno o más caracteres. En esta solución se muestra la manera en que se usan.

Paso a paso

La revisión de un archivo requiere los pasos siguientes:

1. Para obtener pero no eliminar el siguiente carácter del flujo de entrada, llame a **peek()**.
2. Para regresar un carácter al flujo de entrada, llame a **unget()**.
3. Para ignorar caracteres hasta que se encuentra uno específico o hasta que se ha ignorado un número específico de caracteres, llame a **ignore()**.

Análisis

Puede obtener el siguiente carácter en el flujo de entrada sin eliminarlo de ese flujo al usar **peek()**. Tiene este prototipo:

```
int_type peek()
```

Devuelve el siguiente carácter en el flujo o el indicador de final de archivo si se encuentra éste, que es **traits_type::eof()**. El tipo **int_type** es un **typedef** para alguna forma de entero.

Puede devolver el último carácter leído de un flujo empleando **unget()**. Esto permite que el carácter se lea por la siguiente operación de entrada. Aquí se muestra la función **unget()**:

```
istream &unget()
```

Si aún no se han leído caracteres del flujo, ocurre un error y se establece **badbit**. La función devuelve una referencia al flujo.

La función **ignore()** lee y descarta caracteres del flujo de entrada. Tiene este prototipo:

istream &ignore(streamsize *num*=1, int_type *delim* = traits_type::eof())

Lee y descarta caracteres hasta que se han ignorado *num* caracteres (1, como opción predeterminada) o hasta que se encuentra el carácter especificado por *delim*. Como opción predeterminada, *delim* es **traits_type::eof()**. Si se encuentra el carácter delimitador, se elimina del flujo de entrada. Si se encuentra el final del archivo, entonces se establece la marca de estado **eofbit** asociada con el flujo. El tipo **streamsize** es un **typedef** para alguna forma de entero que puede contener el número más grande de bytes que pueden transferirse en cualquier operación de E/S. El tipo **int_type** es un **typedef** para alguna forma de entero. La función devuelve una referencia al flujo.

De las tres funciones, la más interesante es **ignore()** porque le da una manera fácil y eficiente de buscar en un flujo la aparición de un carácter. Una vez que se ha encontrado este archivo, puede empezar a leer (o escribir) el flujo en ese punto. Esto puede ser muy útil en diversas situaciones. Por ejemplo, si tiene un flujo que contiene números de ID de empleado en la forma #dddd (como #2244), entonces puede buscar fácilmente un número de ID al ignorar caracteres hasta que se encuentre un #.

Ejemplo

En el siguiente ejemplo se muestran **peek()**, **unget()** e **ignore()** en acción. El programa crea primero un archivo de datos llamado **prueba.dat** que contiene varios ID de empleado. Sin embargo, hay dos tipos de ID. El primero es un número de cuatro dígitos en la forma #dddd, como #0101. El segundo ID es un marcador de posición que usa una palabra para describir por qué falta el número de ID. Luego el programa busca, lee y despliega todos los ID en el archivo. Para realizar esto, utiliza la revisión de un archivo.

```
// Demuestra peek(), unget() e ignore().
//
// Este programa lee un archivo que contiene dos tipos
// de ID. El primero es un número de cuatro dígitos en esta
// forma: #dddd. El segundo es una palabra que describe
// por qué falta el número de ID. El programa crea un
// archivo de datos llamado prueba.dat que contiene varios
// números de ID. Luego, el programa busca, lee y despliega
// todos los ID del archivo.

#include <iostream>
#include <fstream>
#include <cctype>

using namespace std;

int main()
{
  char car;
  char numid[5];

  // numid termina en un carácter nulo de modo que puede contener una cadena char *.
  numid[4] = 0;

  // Crea un objeto de ofstream y trata de abrir el archivo prueba.dat.
  ofstream archsalida("prueba.dat");

  // Verifica que el archivo se ha abierto correctamente.
  if(!archsalida) {
    cout << "No puede abrir prueba.dat para salida.\n";
    return 1;
  }

  // Escribe alguna información en el archivo.
  archsalida << "Luis Soto #5345\nRafael Romo #negado\nTere Torres #6922\n";
  archsalida << "Hugo Herrera #pendiente\n, Sara Jara, #8875\n";

  // Cierra el archivo de salida.
  archsalida.close();

  if(!archsalida.good()) {
    cout << "Error al crear el archivo de datos.";
    return 1;
  }
```

```
 // Trata de abrir el archivo prueba.dat.
ifstream archentrada("prueba.dat");

if(!archentrada) {
  cout << "No se puede abrir prueba.dat para entrada.\n";
  return 1;
}

// Usa excepciones para revisar errores.
archentrada.exceptions(ios::badbit | ios::failbit);

try {

  // Encuentra y despliega todos los números de ID:
  do {
    // Encuentra el inicio de un número de ID.
    archentrada.ignore(40, '#');

    // Si se encuentra el final del archivo, deja de leer.
    if(archentrada.eof()) {
      archentrada.clear(); // limpia eofbit
      break;
    }

    // Obtiene pero no extrae el siguiente carácter después de #.
    car = archentrada.peek();

    // Ve si el siguiente carácter es un dígito.
    if(isdigit(car)) {

      // Si el carácter es un dígito, lee el número de ID. Como
      // numid tiene un nulo en el quinto carácter, la lectura de
      // cuatro caracteres en los primeros cuatro elementos crea
      // una cadena terminada en un carácter nulo.
      archentrada.read((char *)numid, 4);

      cout << "ID #: " << numid << endl;

    } else {

      // Debido a que el siguiente char no es un dígito, lee la descripción.
      cout << "ID no disponible: ";

      car = archentrada.get();
      while(isalpha(car)) {
        cout << car;
        car = archentrada.get();
      };

      // Regresa el char que no es una letra para que pueda encontrarse
      // y otras instrucciones get() lo procesen.
      archentrada.unget();

      cout << endl;
```

```
      }
    } while(archentrada.good());
  } catch(ios_base::failure exc) {
    cout << "Error al leer el archivo de datos.\n";
  }

  try {
    // Cierra prueba.dat para entrada.
    archentrada.close();
  } catch (ios_base::failure exc) {
    cout << "Error al cerrar el archivo de datos.";
    return 1;
  }

  return 0;

}
```

Aquí se muestra la salida:

```
ID #: 5345
ID no disponible: negado
ID #: 6922
ID no disponible: pendiente
ID #: 8875
```

En este programa se utiliza la revisión de archivo para leer los ID. En primer lugar, se usa **ignore()** para encontrar un carácter #; éste marca el inicio de un ID. Luego se utiliza **peek()** para determinar si lo que sigue es un número de ID real o una descripción verbal. Si el carácter obtenido de **peek()** es un dígito, se lee un número de cuatro dígitos. De otra manera, se lee la descripción. Ésta termina en cuanto se lee un carácter no alfabético. En este caso, el último carácter leído se coloca de nuevo en el flujo de entrada.

Un tema adicional de interés: observe que el programa usa una combinación de excepciones y funciones de detección de errores para buscar éstos. Ésta es una parte de la capacidad del sistema de E/S de C++: puede usar cualquier método que funcione mejor para la situación que se tiene entre manos.

Opciones

Como se explicó, **unget()** devuelve el carácter más recientemente leído al flujo que invoca. Puede "devolver" un carácter diferente de éste al llamar a **putback()**. Aquí se muestra:

istream &putback(char *car*)

Coloca *car* en el flujo para que sea el primer carácter leído por la siguiente operación de entrada. Si ocurre un error, **badbit** se establece en el flujo que invoca.

Otra función que a veces es útil en situaciones de revisión es **readsome()**. En esencia, lee caracteres del búfer de entrada. Si no hay suficientes caracteres en el búfer para satisfacer la solicitud, entonces se establece **eofbit** en el flujo que invoca. Aquí se muestra la función:

streamsize readsome(char **buf*, streamsize *num*)

Trata de leer *num* caracteres del búfer de entrada, almacenándolos en *buf*. Devuelve el número de caracteres que se leyó en realidad.

Otra función que puede ser útil cuando se revisa un archivo (y para muchos otros propósitos) es **gcount()**. Se muestra a continuación:

```
streamsize gcount() const
```

Devuelve el número de caracteres leído por una llamada anterior a una función de entrada sin formato.

Use los flujos de cadena

Componentes clave		
Encabezados	**Clases**	**Funciones**
<sstream>	istringstream ostringstream stringstream	string str() const

Como se explicó en *Revisión general de E/S*, C++ da soporte al uso de una cadena como origen o destino de operaciones de E/S. Para permitir esto, define tres clases de plantilla de flujo de cadena llamadas **basic_istringstream**, **basic_ostringstream** y **basic_stringstream**. Aquí se muestran sus formas de **char**:

istringstream	Usa una **string** para entrada.
ostringstream	Usa una **string** para salida.
stringstream	Usa una **string** para entrada y salida.

En general, las clases de flujo de cadena funcionan como las otras clases de flujo. La única diferencia es que el origen o destino de los datos es una **string** en lugar de algún dispositivo externo. En esta solución se demuestra su uso.

Paso a paso

Para el uso de un flujo de cadena se requieren estos pasos:

1. Cree un flujo de cadena al usar uno de los constructores correspondientes.
2. Realice E/S del flujo de la misma manera en que lo haría empleando cualquier otro tipo de flujo, como uno de archivo.
3. Para obtener el contenido de un búfer de cadena, llame a **str()**.

Análisis

Para crear un flujo de cadena, usará uno de los constructores de flujo de cadena. Cada flujo de cadena define dos constructores, uno que lo inicializa con una cadena y otro que no lo hace. Cuando se realiza entrada, por lo general inicializará la cadena. Para salida, a menudo no necesitará inicializarla. En situaciones de entrada/salida, inicializará la cadena dependiendo de su aplicación.

Aquí se muestra el constructor **istringstream** usado en esta solución:

explicit istringstream(const string &*buf*, ios::openmode *modo* = ios::in).

Crea un flujo de entrada de **char** basado en una cadena. Inicializa esta cadena con el contenido de *buf*. Por tanto, las operaciones de lectura obtendrán los caracteres pasados mediante *buf*.

Aquí se muestra el constructor **ostringstream** usado en esta solución:

explicit ostringstream(ios::openmode *modo* = ios::out)

Crea un flujo de salida de **char** basado en una cadena. Todas las operaciones de escritura pondrán caracteres en una cadena mantenida por **ostringstream**.

Aquí se muestra el constructor **stringstream** usado en esta solución:

explicit stringstream(ios::openmode *modo* = ios::in | ios::out)

Crea un flujo de cadena de **char** que permite entrada y salida. El búfer no está inicializado. Cuando se cambia entre lectura y escritura, por lo general debe realizar una operación de búsqueda o limpieza. (Consulte *Lea un archivo y escriba en él*.)

Puede obtener el contenido actual de la cadena al llamar a esta versión de **str()**:

string str() const

Devuelve una copia del contenido del búfer de cadena actual.

Un tema adicional: no es necesario cerrar un flujo de cadena. En realidad, las clases de flujo de cadena no definen una función **open()** ni **close()**. Esto se debe a que las clases de flujo de cadena no operan sobre un dispositivo externo. Simplemente tratan una cadena como el origen de la entrada o el destino de la salida del flujo. Por esto no es necesario confirmar que un flujo de cadena se creó correctamente antes de usarlo.

Ejemplo

En el siguiente ejemplo se muestran las clases de flujo de cadena en acción.

```
// Usa un flujo de cadena.

#include <iostream>
#include <sstream>

using namespace std;

int main()
{
  char car;

  // Crea un flujo de salida.
  ostringstream cadsalida;
```

```
cout << "Usa un flujo de cadena de salida llamado cadsalida.\n";

// Escribe una salida en el flujo de cadena.
cadsalida << 10 << " " << -20 << " " << 30.2 << "\n";
cadsalida << "Esto es una prueba.";

// Ahora, obtiene una copia del contenido del búfer del flujo
// y lo usa para desplegar el contenido del búfer.
cout << "El contenido actual de cadsalida se obtiene de str():\n"
     << cadsalida.str() << endl;

// Escribe algo más a cadsalida.
cadsalida << "\nSe trata de salida adicional.\n";

cout << endl;

cout << "Se usa un flujo de cadena de entrada llamado cadentrada.\n";

// Ahora, usa el contenido de cadsalida para crear cadentrada:
istringstream cadentrada(cadsalida.str());

// Despliega el contenido de cadentrada mediante llamadas a get().
cout << "El contenido actual de cadentrada mediante get():\n";
do {
  car = cadentrada.get();
  if(!cadentrada.eof()) cout << car;
} while(!cadentrada.eof());

cout << endl;


// Ahora crea el flujo de cadena para entrada/salida.
cout << "Ahora, se usa un flujo de cadena llamado cadentrsal.\n";

stringstream cadentrsal;

// Escribe alguna salida en cadentrsal.
cadentrsal << 10 << " + " << 12 << " is " << 10+12 << endl;

// Ahora, despliega el contenido de cadentrsal mediante get().

cout << "El contenido actual de cadentrsal mediante get():\n";
do {
  car = cadentrsal.get();
  if(!cadentrsal.eof()) cout << car;
} while(!cadentrsal.eof());
cout << endl;

// Limpia eofbit en cadentrsal.
cadentrsal.clear();

cadentrsal << "Salida adicional para cadentrsal.\n";

// Lo siguiente seguirá leyendo desde el punto en que se detuvieron
```

```
  // las lecturas adicionales.
  cout << "Ahora se presentan los caracteres que se acaban de agregar a cadentr-
sal:\n";
  do {
    car = cadentrsal.get();
    if(!cadentrsal.eof()) cout << car;
  } while(!cadentrsal.eof());
}
```

Aquí se muestra la salida:

```
Usa un flujo de cadena de salida llamado cadsalida.
El contenido actual de cadsalida se obtiene de str():
10 -20 30.2
Esto es una prueba.

Usa un flujo de cadena de entrada llamado cadentrada.
El contenido actual de cadentrada mediante get():
10 -20 30.2
Esto es una prueba.
Se trata de salida adicional.

Ahora, usa un flujo de cadena llamado cadentrsal.
El contenido actual de cadentrsal mediante get():
10 + 12 es 22

Ahora se presentan los caracteres que se acaban de agregar a cadentrsal:
Salida adicional para cadentrsal.
```

Opciones

Cuando se crea una instancia de **ostringstream**, es posible inicializar el búfer con una secuencia de caracteres empleando esta versión de su constructor:

explicit ostringstream(const string &*buf*, ios::openmode *modo* = ios::out)

Aquí, el contenido de *buf* se copiará en el búfer de salida.

Cuando se crea una instancia de **istringstream**, no es necesario inicializar el búfer de entrada con una secuencia de caracteres. (Puede establecer el contenido del búfer de flujo de cadena después del hecho al llamar a una segunda forma de **str()**, que se mostrará en breve.) He aquí la versión de **istringstream** que no inicializa el búfer de entrada:

explicit istringstream(ios::openmode *modo* = ios::in)

Observe que sólo se especifica el *modo*, y su opción predeterminada es de entrada.

Para **stringstream**, puede inicializar el búfer con una secuencia conocida de caracteres al usar la forma de su constructor:

explicit stringstream(const string &*buf*, ios::openmode *modo* = ios::in | ios::out)

El contenido de *buf* se copia en el búfer asociado con el objeto **stringstream**.

Para las tres clases de flujo de cadena, puede establecer el contenido del búfer al llamar a esta forma de **str()**:

void str(const string &*buf*)

Reinicializa el búfer con el contenido de *buf*.

Cree insertadores y extractores personalizados

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ostream>	ostream	ostream &operator<<(ostream &flujo, const class_type &*obj*)
<istream>	istream	istream &operator<<(istream &flujo, class_type &*obj*)

En el lenguaje de C++, el operador de salida **<<** es conocido como el *operador de inserción* porque inserta caracteres en un flujo. De igual manera, el operador de entrada **>>** es denominado *operador de extracción* porque extrae caracteres de un flujo. Las funciones que sobrecargan a los operadores de inserción y extracción suelen denominarse *insertadores* y *extractores*, respectivamente. Las clases de E/S de C++ sobrecargan a los operadores de inserción y extracción para todos los tipos integrados. Sin embargo, también es posible crear sus propias versiones sobrecargadas de estos operadores para los tipos de clase que cree. En esta solución se muestra el procedimiento.

Paso a paso

Para sobrecargar un insertador para objetos de clase se necesitan estos pasos:

1. Sobrecargue el operador **<<** para que tome una referencia a un **ostream** en su primer parámetro y una a **const** al objeto para salida en el segundo parámetro.
2. Implemente el insertador para que dé salida al objeto en la manera en que lo desee.
3. Haga que el insertador devuelva la referencia al flujo.
4. Por lo general, hará que el insertador sea un amigo de la clase en que está operando, de modo que tenga acceso a los miembros privados de la clase.

Para sobrecargar un extractor para objetos de clase, se necesitan estos pasos:

1. Sobrecargue el operador **>>** para que tome una referencia a un **istream** en su primer parámetro y una referencia al objeto que recibe entrada en el segundo parámetro.
2. Implemente el extractor para que lea el flujo de entrada y almacene los datos en un objeto de la clase.

3. Haga que el extractor devuelva la referencia al flujo.
4. Por lo general, hará que el extractor sea un amigo de la clase en que está operando, de modo que tenga acceso a los miembros privados de la clase.

Análisis

Es muy simple crear un insertador para una clase que cree. He aquí una forma general típica para un insertador:

```
ostream &operator<<(ostream &flujo, const tipo_clase &obj)
{
    // cuerpo del insertador
    devuelve stream;
}
```

Observe que la función devuelve una referencia a un flujo de tipo **ostream**. Más aún, el primer parámetro a la función es una referencia al flujo de salida. El segundo parámetro es una referencia a **const** al objeto que habrá de insertarse. Técnicamente, el segundo parámetro puede recibir una copia del objeto (es decir, puede ser un parámetro de valor), y no es necesario que sea **const**. Sin embargo, lo más común es que no se altere cuando un objeto es salida, y suele ser más rápido pasarlo por referencia que por valor. Así, por lo general el segundo parámetro es una referencia a **const** para el objeto. Por supuesto, esto está determinado por la situación específica. En todos los casos, el insertador debe devolver *flujo*. Esto permite que el insertador se use en una expresión de E/S más grande.

Dentro de una función de insertador, puede poner cualquier tipo de procedimiento u operación que desee. Es decir, depende por completo de usted la manera en que el insertador dará salida al objeto. Sin embargo, en todos los casos, para que el insertador se mantenga con las buenas prácticas de programación, no debe producir efectos colaterales. Por tanto, no debe modificarse el objeto. Tampoco debe realizar operaciones que no estén relacionadas con la inserción. Por ejemplo, ¡tal vez no sea buena idea hacer que un insertador recicle la memoria no utilizada como efecto colateral a una operación de inserción!

Los extractores son el complemento de los insertadores. Almacenan entrada en un objeto. La forma general de una función extractora es:

```
istream &operator>>(istream &flujo, const tipo_clase &obj)
{
    // cuerpo del extractor
    devuelve stream;
}
```

Los extractores devuelven una referencia a un flujo de tipo **istream**, que es un flujo de entrada. El primer parámetro también debe ser una referencia a un flujo de tipo **istream**. Observe que el segundo parámetro debe ser una referencia a un objeto de la clase para la que el extractor está sobrecargado. Esto es así para que el objeto pueda modificarse mediante la operación de entrada (extracción).

Como los insertadores, un extractor debe confinar sus operaciones para leer datos del flujo de entrada y almacenarlo en el objeto especificado. No debe generar efectos colaterales. No debe leer más entrada que necesaria para el objeto. Por ejemplo, un extractor por lo general no debe leer un espacio final.

En muchos casos, querrá hacer que el insertador o el extractor sea un amigo de la clase para la que está sobrecargado. Al hacerlo así, otorga acceso a los miembros privados de la

clase. Esto podría requerirse para obtener datos para salida o para almacenar datos de entrada. Por supuesto, esto no sería posible si estuviera creando un insertador o extractor para una clase a la que no tiene el código fuente, como una clase de terceros.

Ejemplo

A continuación se muestran ejemplos de un insertador y un extractor personalizados. Crea una clase llamada **TresD**, que almacena coordenadas tridimensionales. Utiliza un insertador personalizado para dar salida a las coordenadas. Utiliza un extractor personalizado para leer las coordenadas.

```
// Demuestra un insertador y extractor de objetos
// de tipo TresD.

#include <iostream>

using namespace std;

class TresD {
  int x, y, z; // Coordenadas 3-D
public:
  TresD(int a, int b, int c) { x = a; y = b; z = c; }

  // Hace que el insertador y el extractor sean amigos de TresD.
  friend ostream &operator<<(ostream &flujo, const TresD &obj);
  friend istream &operator>>(istream &flujo, TresD &obj);

  // ...
};

// Insertador TresD. Despliega las coordenadas X, Y, Z.
ostream &operator<<(ostream &flujo, const TresD &obj)
{
  flujo << obj.x << ", ";
  flujo << obj.y << ", ";
  flujo << obj.z << "\n";
  return flujo;  // devuelve el flujo
}

// Extractor TresD. Obtiene valores tridimensionales.
istream &operator>>(istream &flujo, TresD &obj)
{
  flujo >> obj.x >> obj.y >> obj.z;
  return flujo;
}

int main()
{
  TresD td(1, 2, 3);

  cout << "Las coordenadas en td: " << td << endl;

  cout << "Ingrese las nuevas coordenadas 3D: ";
  cin >> td;
```

```
  cout << "Las coordenadas en td son ahora: " << td << endl;

  return 0;
}
```

Aquí se muestra una ejecución de ejemplo:

```
Las coordenadas en td: 1, 2, 3

Ingrese las nuevas coordenadas 3D: 9 8 7
Las coordenadas en td son ahora: 9, 8, 7
```

Opciones

Como se mencionó, cuando se crea un insertador, no es técnicamente necesario pasar por referencia el objeto al que se está dando salida. En algunos casos, tal vez quiera usar, en cambio, un parámetro de valor. Esto podría tener sentido cuando se opera sobre objetos muy pequeños en que la cantidad de tiempo que se requiere para sacar el objeto de una pila (que es lo que sucede cuando se pasa un argumento por valor) es menor que la que toma extraer la dirección del objeto (que es lo que sucede cuando un objeto se pasa por referencia).

Cree un manipulador sin parámetros

Componentes clave		
Encabezados	**Clases**	**Funciones**
<istream>	istream	istream &*nombre-manip*(istream &*flujo*)
<ostream>	ostream	ostream &*nombre-manip*(ostream &*flujo*)

Los manipuladores de E/S son funciones que están insertadas dentro de una expresión de E/S. Afectan el flujo, como cuando cambian sus marcas de formato, o insertan caracteres en un flujo o los extraen de él. Debido a que operan dentro de una expresión de E/S, los manipuladores afinan la codificación de muchas tareas. C++ proporciona muchos manipuladores integrados, y se describen en el capítulo 6, donde se presentan las soluciones relacionadas con la formación de datos. Sin embargo, también es posible crear sus propios manipuladores personalizados.

Por lo general, se usa un manipulador personalizado para consolidar una secuencia o separar operaciones de E/S en un solo paso. Por ejemplo, no es poco común que tengan situaciones en que la misma secuencia de operaciones de E/S ocurre con frecuencia dentro de un programa. En esos casos, puede usar un manipulador personalizado para realizar estas acciones, con lo que simplifica su código fuente y se evitan errores. He aquí otro ejemplo: tal vez necesite realizar operaciones de E/S en un dispositivo que no es estándar. Por ejemplo, podría usar un manipulador para enviar códigos de control a un tipo especial de impresora o a un sistema de reconocimiento óptico. Un manipulador personalizado puede simplificar este proceso al permitirle que envíe los códigos por nombre. Cualesquiera que sean los propósitos, los manipuladores personalizados son extensiones populares del sistema de E/S de C++.

Hay dos tipos básicos de manipuladores: los que operan en los flujos de entrada y los que lo hacen en los de salida. Además de estas dos amplias categorías, hay una división secundaria: los manipuladores que toman un argumento y los que no. Las técnicas usadas para crear manipuladores sin parámetros difieren de las usadas para crear otros con parámetros. En esta solución se muestra cómo crear manipuladores personalizados sin parámetros. En la siguiente solución se muestra una manera de crear manipuladores con parámetros.

Paso a paso

Para crear su propio manipulador de salida sin parámetros se requieren estos pasos:

1. Cree una función que tome una referencia a un objeto de **ostream** como un parámetro y devuelva una referencia a un **ostream**.
2. Dentro de esa función, realice acciones en el **ostream** pasado como argumento.
3. Devuelva una referencia al argumento de **ostream**.

Para crear su propio manipulador de entrada sin parámetros se requieren estos pasos:

1. Cree una función que tome una referencia a un objeto de **istream** como un parámetro y devuelva una referencia a un **istream**.
2. Dentro de esa función, realice acciones en el **istream** pasado como argumento.
3. Devuelva una referencia al argumento de **istream**.

Análisis

Todas las funciones de manipulador de salida sin parámetros tienen este esqueleto:

```
ostream &nombre-manip(ostream &flujo)
{
  // aquí va su código
  return flujo;
}
```

Aquí, *nombre-manip* es el nombre del manipulador y *flujo* es una referencia al flujo de salida en que operará el manipulador. Observe que también se devuelve *flujo*. Esto es necesario para permitir que el manipulador se use como parte una expresión de E/S más larga. Es importante tomar nota de que aunque el manipulador tenga como único argumento una referencia al flujo en que está operando, no se usa un argumento cuando el manipulador se inserta en una operación de salida.

Todas las funciones de manipulador de entrada sin parámetros tienen este esqueleto:

```
istream &nombre-manip(istream &flujo)
{
  // aquí va su código
  return flujo;
}
```

Un manipulador de entrada recibe una referencia al flujo para el que se invocó. El manipulador debe devolver este flujo. Aunque éste toma un argumento de **istream**, no se pasan argumentos cuando se invoca el manipulador.

Una vez que haya definido un manipulador, puede usarlo con sólo especificar su nombre en una expresión de inserción o extracción. La razón por la que esto funciona es que los operadores **>>** y **<<** se sobrecargan para aceptar un apuntador a función que tiene una referencia a flujo como único parámetro. Los operadores **<<** y **>>** se implementan de modo que pueden llamar a la función mediante el apuntador, pasando en una referencia al flujo. Este proceso le permite que su manipulador personalizado reciba una referencia al flujo que se afectará.

Es importante comprender que (excepto en casos muy inusuales) su manipulador debe operar en el flujo que se le pasa. Un error común que cometen los principiantes consiste en incluir en el código una referencia al flujo, como **cout**, en lugar de usar el flujo pasado al parámetro. El problema es que su manipulador funcionará correctamente en algunos casos y fallará en otros. Aunque este error suele ser fácil de encontrar y corregir, en ocasiones es intimidante, dependiendo del flujo en que lo haya codificado. La regla es fácil: un manipulador debe operar en el flujo que se pasa.

Ejemplo

En el siguiente ejemplo se muestra un manipulador personalizado de entrada y salida. Al manipulador de salida se le llama **relleno_ast()**. Especifica el asterisco (*) como carácter de relleno y asigna 10 al ancho de campo. Por tanto, después de una llamada a **relleno_ast()**, se despliega el número 1234 como ******1234. (Para conocer más acerca de la formación de datos, consulte el capítulo 6.) El manipulador de entrada se denomina **omitir_digitos().** Omite los dígitos iniciales en el flujo de entrada. Por tanto, si el flujo de entrada contiene 9786ABC0101, entonces lee y descarta el 9786 inicial y deja ABC0101 en el flujo de entrada.

```
// Demuestra un manipulador de salida personalizado llamado relleno_ast()
// y un manipulador de entrada personalizado de nombre omitir_digitos().

#include <iostream>
#include <iomanip>
#include <string>
#include <cctype>

using namespace std;

// Un manipulador de salida simple que establece el carácter de relleno
// como * y establece el ancho de campo en 10.
ostream &relleno_ast(ostream &flujo) {

  flujo << setfill('*') << setw(10);

  return flujo;
}

// Un manipulador de entrada simple que omite los dígitos iniciales.
istream &omitir_digitos(istream &flujo) {
  char car;

  do {
    car = flujo.get();
```

```
  } while(!flujo.eof() && isdigit(car));
  if(!flujo.eof()) flujo.unget();

  return flujo;
}

int main()
{
  string cad;

  // Demuestra el manipulador de salida personalizado.
  cout << 512 << endl;
  cout << relleno_ast << 512 << endl;

  // Demuestra el manipulador de entrada personalizado.
  cout << "Ingrese algunos caracteres: ";
  cin >> omitir_digitos >> cad;
  cout << "Contenido de cad: " << cad;

  return 0;
}
```

He aquí una ejecución de ejemplo:

```
512
*******512
Ingrese algunos caracteres: 123ABC
Contenido de cad: ABC
```

Opciones

Si ha codificado correctamente su manipulador personalizado para que opere en el flujo que se ha pasado, entonces puede usarse en cualquier tipo de flujo. Por ejemplo, en el programa anterior, puede usar **relleno_ast()** en un flujo de archivo o uno de cadena. Para confirmar esto, agregue la siguiente secuencia al programa. Utiliza **relleno_ast()** en un **ostringstream** y un **ofstream**.

```
// Usa relleno_ast () en un stringstream.
ostringstream flujosalida;
flujosalida << relleno_ast << 29;
cout << flujosalida.str();

// Usa relleno_ast en un ofstream.
ofstream archsalida("prueba.dat");
if(!archsalida) {
  cout << "Error al abrir el archivo.\n";
  return 1;
}
archsalida << relleno_ast << 19;
```

Después de volver a compilar, verá que **relleno_ast()** funciona correctamente en **flujosalida** y **archsalida**.

También puede crear manipuladores personalizados. El proceso es el tema de la siguiente solución.

Cree un manipulador con parámetros

Componentes clave		
Encabezados	**Clases**	**Funciones y campos**
<istream>	istream	istream &operator>>(istream &*flujo*, *clase-manip cm*)
<ostream>	ostream	ostream &operator<<(ostream &*flujo*, *clase-manip cm*)
	clase-manip	definido por el usuario

Como se mostró en la solución anterior, es muy fácil crear un manipulador sin parámetros. La razón es que **<<** o **>>** están sobrecargados para (entre muchas otras cosas) un apuntador a función. Como se explicó en la solución anterior, cuando se usa un manipulador sin parámetros, se pasa un apuntador al insertador o extractor sobrecargado y se llama a la función, y el flujo se pasa como argumento. Por desgracia, este mecanismo simple no funcionará con manipuladores que requieren un argumento porque no hay manera de pasar un argumento mediante el apuntador a función. Como resultado, la creación de un manipulador con parámetros depende de un mecanismo fundamentalmente diferente, que es un poco más complicado. Más aún, hay varias maneras de implementar un manipulador con parámetros. En esta solución se muestra una manera relativamente simple y sencilla.

Paso a paso

Para crear un manipulador de salida con parámetros se necesitan estos pasos:

1. Cree una clase cuyo nombre sea el del manipulador. Por ejemplo, si éste se llama **mimanip**, entonces el nombre de la clase debe ser **mimanip**.
2. Cree un campo privado en la clase que contendrá el argumento pasado al manipulador. El tipo del campo debe ser el mismo que el tipo de datos que se pasará al manipulador.
3. Cree un constructor para la clase que tenga un parámetro, que sea del mismo tipo que el de los datos que se pasarán al manipulador. Haga que el constructor inicialice el valor del campo del paso 2 con el pasado al constructor.
4. Cree un insertador sobrecargado que tome una referencia a **ostream** como primer argumento y un objeto de la clase del paso 1 como su segundo argumento. Dentro de esta función, realice las acciones del manipulador. Devuelva una referencia al flujo.
5. Haga que el insertador sobrecargado sea un amigo de la clase del paso 1.
6. Para usar el manipulador, use el constructor de la clase en la expresión de salida, pasándolo en el argumento deseado. Esto causará que se construya un argumento, y luego se llamará al insertador sobrecargado, empleando ese objeto como operando del lado derecho.

Para crear un manipulador de entrada con parámetros, se necesitan estos pasos:

1. Cree una clase cuyo nombre sea el del manipulador. Por ejemplo, si éste se llama **mimanip**, entonces el nombre de la clase debe ser **mimanip**.
2. Cree un campo privado en la clase que contendrá el argumento pasado al manipulador. El tipo del campo debe ser el mismo que el tipo de datos que se pasará al manipulador.
3. Cree un constructor para la clase que tenga un parámetro, que sea del mismo tipo que el de los datos que se pasarán al manipulador. Haga que el constructor inicialice el valor del campo del paso 2 con el pasado al constructor.
4. Cree un extractor sobrecargado que tome una referencia a **istream** como primer argumento y un objeto de la clase del paso 1 como su segundo argumento. Dentro de esta función, realice las acciones del manipulador. Devuelva una referencia al flujo.
5. Haga que el extractor sobrecargado sea un amigo de la clase del paso 1.
6. Para usar el manipulador, use el constructor de la clase en la expresión de entrada, pasándolo en el argumento deseado. Esto causará que se construya un argumento, y luego se llamará al extractor sobrecargado, empleando ese objeto como operando del lado derecho.

Análisis

En general, la creación de un manipulador con parámetros requiere dos elementos. El primero es una clase que almacene el argumento pasado al manipulador. El segundo es un insertador o extractor que esté sobrecargado para tomar un objeto de esa clase como operando del lado derecho. Cuando el manipulador se incluye en una expresión de E/S, se construye un objeto de la clase, y el argumento se guarda. Luego el insertador o extractor opera en ese objeto y puede acceder al argumento.

Trabajemos esto paso a paso, creando un insertador con parámetros simple llamado **sangrado**, que da sangría a la salida con un número específico de espacios. Por ejemplo, la expresión

```
cout << sangrado(10) << "Hola";
```

causará que se dé salida a 10 espacios, seguidos por la cadena "Hola". Como se explicó, todos los manipuladores con parámetros requieren dos elementos. El primero es una clase que almacena el argumento pasado al manipulador. Por tanto, para crear el manipulador **sangrado**, empiece por crear una clase llamada **sangrado** que almacene el argumento pasado a su constructor y especifique un insertador sobrecargado como amigo, como se muestra aquí:

```
// Una clase que da soporte al manipulador de salida sangrado.
class sangrado {
  int len;
public:
  sangrado(int i) { len = i; }
  friend ostream &operator<<(ostream &flujo, sangrado ndt);
};
```

Como puede ver, el constructor toma un argumento, que se almacena en el campo privado **len**. Ésta es la única funcionalidad que proporciona **indent**. Simplemente almacena el argumento. Sin embargo, declara que **operator<<()** es un amigo. Esto le da a la función de operador acceso al campo privado **len**.

El segundo elemento que necesita crear es un insertador sobrecargado que toma una instancia de **sangrado** como operando del lado derecho. (Consulte *Cree insertadores y extractores personalizados* para conocer detalles sobre la creación de un insertador o un extractor.) Haga que este operador dé salida al número de espacios especificado por el campo **len** del objeto en que está operando. He aquí una manera de implementar esta función:

```
// Crea un insertador para objetos de tipo sangrado.
ostream &operator<<(ostream &flujo, sangrado ndt) {

  for(int i=0; i <  ndt.len; ++i) flujo << " ";

  return flujo;
}
```

Como puede ver, este operador toma una referencia a **ostream** como operando del lado izquierdo y un objeto de **sangrado** como operando del lado derecho. Da salida el número de espacios especificado por el objeto de **sangrado** y luego devuelve el flujo. Debido a que **operator<<()** es un amigo de **sangrado**, puede acceder al campo **len**, aunque sea privado.

Cuando se usa **sangrado** dentro de una expresión de salida, causa que un objeto de tipo **sangrado** se cree con el argumento especificado. Luego, se invoca la función **operator<<()** sobrecargada, pasándola en el flujo y en el objeto de **sangrado** recién creado.

Ejemplo

En el siguiente ejemplo se muestran un manipulador de entrada y uno de salida con parámetros. Al manipulador de entrada se le denomina **omitircar** y, en la entrada, omite los caracteres iniciales que coinciden con el pasado a **omitircar**. Por ejemplo, **omitircar('X')** omite las X al principio. El manipulador de salida es **sangrado**, que se describió en la secuencia *Análisis* de esta solución.

```
// Crea manipuladores simples de entrada y salida con parámetros.
//
// El manipulador sangrado da salida un número específico de espacios.
// El manipulador omitircar omite un carácter específico en la entrada.

#include <iostream>
#include <string>
#include <sstream>

using namespace std;

// Juntos, la clase y el operador sobrecargado siguientes crean
// el manipulador sangrado.

// Una clase que da soporte al manipulador de salida sangrado.
class sangrado {
  int len;
public:
  sangrado(int i) { len = i; }
  friend ostream &operator<<(ostream &flujo, sangrado ndt);
};

// Crea un insertador para objetos de tipo sangrado.
```

```
ostream &operator<<(ostream &flujo, sangrado ndt) {

  for(int i=0; i <  ndt.len; ++i) flujo << " ";

  return flujo;
}

// Juntos, la clase y el operador sobrecargado siguientes crean
// el manipulador omitircar.

// Una clase que da soporte al manipulador de entrada omitircar.
class omitircar {
  char car;
public:
  omitircar(char c) { car = c; }
  friend istream &operator>>(istream &flujo, omitircar sc);
};

// Crea un extractor para objetos de tipo omitircar.
istream &operator>>(istream &flujo, omitircar sc) {
  char car;

  do {
    car = flujo.get();
  } while(!flujo.eof() && car == sc.car);
  if(!flujo.eof()) flujo.unget();

  return flujo;
}

// Demuestra sangrado y omitircar.
int main() {
  string cad;

  // Usa sangrado para añadir sangrías a la salida.
  cout << sangrado(9) << "Esto se ha sangrado 9 lugares.\n"
       << sangrado(9) << "Igual que esto.\n" << sangrado(18)
       << "Pero esto se ha sangrado 18 lugares.\n\n";

  // Usa omitircar para ignorar los ceros iniciales.
  cout << "Ingresa algunos caracteres: ";
  cin >> omitircar('0') >> cad;
  cout << "Se omiten los ceros iniciales. Contenido de cad: "
       << cad << "\n\n";

  // Usa sangría en un ostringstream.
  cout << "Usa sangrado con un flujo de cadena.\n";
  ostringstream flujocadsal;
  flujocadsal << sangrado(5) << 128;
  cout << "El contenido de flujocadsal:\n" << flujocadsal.str() << endl;

  return 0;
}
```

Aquí se muestra una ejecución de ejemplo:

```
        Esto se ha sangrado 9 lugares.
        Igual que esto.
                         Pero esto se ha sangrado 18 lugares.

Ingresa algunos caracteres: 000abc
Se omiten los ceros iniciales. Contenido de cad: abc

Usa sangrado con un flujo de cadena.
El contenido de flujocadsal:
     128
```

Opciones

En esta solución se muestra una manera fácil de crear manipuladores con parámetros, pero no es la única manera. En el encabezado **<iomanip>** están definidos los manipuladores con parámetros especificados por el estándar C++. Si examina este encabezado, probablemente verá un método más sofisticado, que utiliza plantillas y tal vez macros complejas. Podrá usar el método mostrado en ese encabezado para crear sus propios manipuladores con parámetros que se integren con los tipos de clase definidos por ese encabezado. Sin embargo, debido a que las clases en **<iomanip>** son específicas de la implementación, pueden ser diferentes (y probablemente lo serán) entre compiladores. El método utilizado en esta solución es transportable. Además, por lo general el mecanismo empleado por **<iomanip>** es muy complicado y puede resultar difícil de comprender sin un estudio considerable. A menudo, simplemente es más fácil usar la técnica mostrada en esta solución. Con toda franqueza, es el método preferido por el autor.

Obtenga o establezca una configuración regional y de idioma de flujo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>	ios_base	locale getloc() const
<ios>	ios	locale imbue(const locale &*nuevaubi*)
<locale>	locale	string name() const

Tiene la opción de obtener o establecer el objeto de **locale** asociado con un flujo. En C++, la información específica de la configuración regional está encapsulada dentro de un objeto de **locale**. Este objeto define varios elementos relacionados con la configuración regional y de idioma, como el símbolo monetario, el separador de miles, etc. Cada flujo tiene una configuración asociada. Para ayudar a la internacionalización, tal vez quiera obtener un objeto de **locale** de un flujo, o establecer uno nuevo. En esta solución se muestra el proceso.

Paso a paso

Para obtener el objeto de **locale** actual relacionado con un flujo, se necesitan estos pasos:

1. Cree una instancia de **locale** que recibirá una copia de la configuración regional y de idioma actual.
2. Llame a **getloc()** en el flujo, para obtener una copia de la configuración actual.

Para establecer la configuración regional y de idioma con un flujo, se necesitan estos pasos:

1. Cree una instancia de **locale** que encapsule la configuración regional y de idioma actual.
2. Llame a **imbue()** en el flujo, pasándole el objeto de **locale** del paso 1.

Análisis

La clase **locale** encapsula información geopolítica acerca del entorno de ejecución de un programa. Por ejemplo, la configuración regional y de idioma de un programa determina el símbolo monetario, el formato de hora y de fecha, entre muchos otros. La clase **locale** necesita el encabezado **<locale>**. Cada flujo tiene un objeto de **locale** asociado.

Para obtener la configuración regional actual de un flujo, llame a **getloc()** en el flujo. Se muestra aquí:

locale getloc() const

Devuelve el objeto de **locale** asociado con el flujo.

Para establecer la configuración de un flujo, llame a **imbue()** en el flujo. Aquí se muestra:

locale imbue(const locale &*nuevaubi*)

La configuración regional y de idioma del flujo que invoca se establece en *nuevaubi*, y se devuelve la anterior.

Una manera fácil de construir una instancia de **locale** consiste en usar este constructor:

explicit locale(const char **nombre*)

Aquí, *nombre* especifica el nombre de la configuración, como german, spanish_spain o US. Si *nombre* no representa una configuración regional y de idioma válida, entonces se lanza una excepción **runtime_error**.

Dada una instancia de **locale**, puede obtener su nombre al llamar a **name()**. Aquí se muestra:

string name() const

Se devuelve el nombre legible para el ser humano de la configuración regional y de idioma.

Ejemplo

En el siguiente ejemplo se muestra cómo obtener y establecer una configuración regional de flujo. Primero se despliega la configuración actual del flujo, que suele ser la de C (que, por lo general, es la cadena predeterminada para un programa de C++). Luego establece la configuración en German_Germany. Por último, obtiene y despliega el símbolo monetario y el carácter usado para el separador de miles.

```
// Demuestra getloc() e imbue() en un flujo.

#include <iostream>
```

```
#include <fstream>
#include <locale>

using namespace std;

int main()
{
  ofstream archsalida("prueba.dat");

  if(!archsalida) {
    cout << "No se puede abrir el archivo.\n";
    return 1;
  }

  // Despliega el nombre de la configuración regional y de idioma actual.
  cout << "La configuraci\u00a2n regional inicial es " << archsalida.getloc().name();
  cout << "\n\n";

  cout << "Estableciendo la configuraci\u00a2n regional en German_Germany.\n";

  // Crea un objeto de locale para Alemania.
  locale loc("German_Germany");

  // Establece la configuración regional de archsalida en loc.
  archsalida.imbue(loc);

  // Despliega el nombre de la nueva configuración regional y de idioma.
  cout << "La configuraci\u00a2n regional inicial es ahora " << archsalida.
  getloc().name();
  cout << endl;

  // Primero, confirma que la faceta moneypunct está disponible.
  if(has_facet<moneypunct<char, true> >(archsalida.getloc())) {
    // Obtiene la faceta moneypunct.
    const moneypunct<char, true> &mp =
      use_facet<moneypunct<char, true> >(archsalida.getloc());

    // Despliega el símbolo monetario y el separador de miles.
    cout << "S\u00a1mbolo monetario: " << mp.curr_symbol() << endl;
    cout << "Separador de miles: " << mp.thousands_sep() << endl;
  }

  archsalida.close();

  if(!archsalida.good()) {
    cout << "Error al cerrar el archivo.\n";
    return 1;
  }

  return 0;
}
```

Aquí se muestra la salida:

```
La configuración regional inicial es C

Estableciendo la configuración regional en German_Germany.
La configuración regional inicial es ahora German_Germany.1252
Símbolo monetario: EUR
Separador de miles: .
```

Opciones

Como se mencionó, en el núcleo de la internacionalización se encuentra la clase **locale.** Ésta encapsula un conjunto de *facetas* que describen los aspectos geopolíticos del entorno de ejecución. Las facetas están representadas por clases declaradas dentro de **<locale>**, como **moneypunct** que se usa en el ejemplo. Entre otras, se incluyen **numpunct**, **num_get**, **num_put**, **time_get** y **time_put**. Puede usar estas clases para leer y escribir información que está formada de manera relacionada con una configuración regional y de idioma. Consulte el capítulo 6 para conocer soluciones relacionadas con la formación de datos.*

Use el sistema de archivos de C

Componentes clave		
Encabezados	**Clases**	**Funciones**
<cstdio>		int fclose(FILE *aa) int feof(FILE *aa) int ferror(FILE *aa) FILE *fopen(const char *nombrearch, const char *modo) int fgetc(FILE *aa) int fputc(int car, FILE *aa)

En las soluciones anteriores se ha descrito cómo realizar una amplia variedad de tareas de manejo de archivos al emplear el sistema de E/S de C++, que está basado en la jerarquía de clases descrita en la revisión general presentada al principio de este capítulo. Se trata del sistema de E/S que, por lo general, usará cuando escriba código de C++. Una vez dicho esto, ningún libro de C++ estaría completo sin una solución, por lo menos, que describa los fundamentos del uso del "otro sistema de E/S" de C++, que es el heredado de C.

Como casi todos los programadores de C++ lo saben, C++ se construyó a partir del lenguaje C. Como resultado, C++ incluye todo el lenguaje C. Por eso es por lo que el bucle **for** en C, por ejemplo, funciona igual que lo hace en C++. También es por eso por lo que las funciones basadas en C, como **tolower()**, están disponibles para uso en un programa de C++. Esto es importante porque C define un sistema de E/S completo propio, que está separado del definido por C++. Probablemente ya ha visto al E/S de C en acción en código de terceros. Por ejemplo, la función de salida de la consola principal es **printf()** y una función de uso común para entrada es **scanf()**. En

***Nota del T.** También vale la pena que explore la configuración regional relacionada con su país, para conocer la manera de utilizar ésta con el fin de desplegar caracteres específicos del español.

realidad, variantes de estas funciones se utilizan en algunas de las soluciones del capítulo 6, donde se describe la formación de datos.

Debido a que el sistema de archivos de C tiene soporte completo en C++, en ocasiones verá que se usa en programas de éste. Tal vez más importante sea que gran parte del código heredado de C aún tiene un amplio uso. Si le estará dando mantenimiento a ese tipo de código, o tal vez actualizándolo al sistema de E/S de C++, entonces es necesario un conocimiento básico del funcionamiento del sistema de archivos de C. Por último, en realidad nadie puede llamarse a sí mismo un programador en C++ sin tener por lo menos un poco de conocimientos del subconjunto del lenguaje C, incluido su tratamiento de E/S.

En esta solución se demuestra el mecanismo básico necesario para abrir, cerrar, leer y escribir un archivo. También se muestra la manera de detectar errores. Aunque se podrían presentar muchos elementos más del E/S de archivo de C en una solución, esto le proporcionará una comprensión general de los temas clave.

Paso a paso

Para usar el sistema de E/S de C para leer un archivo y escribir en él, se requieren estos pasos:

1. Abra un archivo al llamar a **fopen()**.
2. Confirme que el archivo está abierto al probar el valor devuelto por **fopen()**. Si es NULL, el archivo no está abierto.
3. Si el archivo está abierto para entrada, lea caracteres al llamar a **fgetc()**.
4. Si el archivo está abierto para salida, lea caracteres al llamar a **fputc()**.
5. Cierre el archivo al llamar a **fclose()**.
6. Revise errores al llamar a **ferror()**.
7. Revise si se alcanzó el final del archivo al llamar a **feof()**.

Análisis

Aunque el sistema de archivos de C utiliza el mismo concepto de alto nivel del flujo, la manera en que funciona es sustancialmente diferente del sistema de archivos de C++. Una diferencia clave es que las funciones de E/S de C operan mediante *apuntadores a archivos*, en lugar de hacerlo sobre objetos de clases que encapsulan un archivo. (Como se explicó, el apuntador a un archivo representa un archivo.) Por tanto, el sistema de archivos de C no se centra en una jerarquía de clases sino alrededor del apuntador a archivos.

Un apuntador a archivos se obtiene al abrir un archivo. Una vez que tenga uno, puede operar en él mediante una o más de las funciones de E/S de C. Aquí se muestran las usadas por esta solución. Todas requieren el encabezado **<cstdio>**. Se trata de la versión de C++ del archivo de encabezado original **stdio.h** usado por C.

Nombre	Función
fopen()	Abre un archivo.
fclose()	Cierra un archivo.
fputc()	Escribe un carácter en un archivo.
fgetc()	Lee un carácter del archivo.
feof()	Devuelve true si se llega al final del archivo.
ferror()	Devuelve true si ha ocurrido un error.

El encabezado **<cstdio>** proporciona los prototipos para las funciones de E/S y define estos tres tipos: **size_t**, **fpos_t** y **FILE**. El tipo **size_t** es alguna variedad de entero sin signo, al igual que **fpos_t**. El tipo **FILE** describe un archivo. Merece mayor atención.

El apuntador a archivo es el subproceso común que une a los procesos de E/S de C. Es un apuntador a una estructura de tipo **FILE**. Esta estructura contiene información que define varios elementos del archivo, incluidos su nombre, estado y la posición actual del archivo. En esencia, el apuntador a archivos identifica un archivo específico, y el flujo asociado lo usa para dirigir la operación de las funciones de E/S. Con el fin de leer o escribir archivos, su programa necesita usar apuntadores a archivo. Para obtener una variable de apuntador a archivo, use una instrucción como ésta:

```
FILE *af;
```

También hay varias macros definidas en **<cstdio>**. Las relevantes para esta solución son **NULL** y **EOF**. La macro **NULL** define un apuntador nulo. La **EOF** suele definirse como –1 y es el valor devuelto cuando una función de entrada trata de leer después del final del archivo.

A continuación se muestra una revisión general de cada función de E/S de C usada en esta solución.

fopen()

La función **fopen()** abre un flujo para su uso y vincula un archivo con ese flujo. Luego devuelve el apuntador a archivo asociado con ese archivo. Con mayor frecuencia (y para el resto del análisis) el archivo es de disco. La función **fopen()** tiene este prototipo:

FILE *fopen(const char **nombrear*, const char **modo*)

donde *nombrear* es un apuntador a una cadena de caracteres que integra un nombre de archivo válido y puede incluir una especificación de ruta. La cadena señalada por *modo* determina la manera en que el archivo se abrirá. En la siguiente tabla se muestran los valores legales para *modo*. (Cadenas como "r+b" también pueden representarse como "rb+".)

Modo	Significado
r	Abre un archivo de texto para lectura.
w	Abre un archivo de texto para escritura.
a	Adjunta a un archivo de texto.
rb	Abre un archivo binario para lectura.
wb	Crea un archivo binario para escritura.
ab	Adjunta a un archivo binario.
r+	Abre un archivo de texto para lectura/escritura.
w+	Crea un archivo de texto para lectura/escritura.
a+	Adjunta o crea un archivo de texto para lectura/escritura.
r+b	Abre un archivo binario para lectura/escritura.
w+b	Crea un archivo binario para lectura/escritura.
a+b	Adjunta o crea un archivo binario para lectura/escritura.

Observe que un archivo puede abrirse en modo de texto o binario. En casi todas las implementaciones, en modo de texto, la secuencia retorno de carro/avance de línea se traduce en caracteres de nueva línea en la entrada. En la salida, ocurre lo inverso: las nuevas líneas se traducen en secuencias retorno de carro/avance de línea. Esta traducción no ocurre en archivos binarios.

Como se estableció, la función **fopen()** devuelve un apuntador a archivo. Su programa nunca debe modificar el valor de este apuntador. Si ocurre un error cuando trata de abrir el archivo, **fopen()** devuelve un apuntador nulo. Debe confirmar que el archivo se abrió con éxito al probar el valor devuelto por **fopen()**. He aquí un ejemplo de la manera en que se abre un archivo con **fopen()**. Trata de abrir un archivo llamado **prueba.dat** para salida.

```
FILE *aa;

if((aa = fopen("prueba.dat", "w"))==NULL) {
  cout << "No se puede abrir prueba.dat para salida.\n";
  exit(1);
}
```

Si el archivo no se puede abrir por alguna razón (por ejemplo, si es de sólo lectura), entonces la llamada a **fopen()** fallará y se devolverá un apuntador nulo. Por supuesto, la prueba para revisar si falla una apertura puede escribirse de manera más compacta, como se muestra a continuación:

```
if (!(aa = fopen("prueba.dat", "w"))) { // ...
```

La prueba explícita contra **NULL** no es necesaria porque un apuntador nulo es un valor falso.

fclose()

La función **fclose()** cierra un flujo que estaba abierto al llamar a **fopen()**. Escribe cualquier dato que sobre en el búfer del disco en el archivo y cierra éste en el nivel formal del sistema operativo. La falla en el cierre de un flujo puede provocar problemas, como datos perdidos, archivos destruidos y posibles errores intermitentes en su programa. Por tanto, siempre debe cerrar un archivo cuando haya terminado con él. Cerrar un archivo también libera cualquier recurso del sistema usado por el archivo, haciéndolo disponible para nuevo uso.

La función **fclose()** tiene este prototipo:

int fclose(FILE **aa*)

donde *aa* es el apuntador a archivo devuelto por la llamada a **fopen()**. Un valor devuelto de cero significa una operación de cierre que ha tenido éxito. La función devuelve **EOF** si ocurre un error. Una llamada a **fclose()** fallará cuando se ha eliminado prematuramente un disco de la unidad o no hay más espacio en el disco, por ejemplo.

fputc()

La función **fputc()** escribe caracteres en un archivo. Aquí se muestra:

int fputc(int *car*, FILE **aa*)

El parámetro *aa* especifica el archivo en que se escribirá, y *car* es el carácter que se escribe. Aunque *car* está definido como **int**, sólo se escribe un byte de orden bajo. Si **fputc()** tiene éxito, devuelve *car*. De otra manera, devuelve **EOF**.

fgetc()

La función **fgetc()** lee caracteres de un archivo. Aquí se muestra:

int fgetc(FILE **aa*)

El parámetro *aa* especifica el archivo que se leerá. Devuelve el siguiente carácter en el archivo, devuelto como un valor **int**. Devuelve **EOF** cuando se ha alcanzado el final del archivo. Por tanto, para leer en el final de un archivo de texto, podría usar el siguiente código:

```
do {
  car = fgetc(aa);
} while(car != EOF);
```

Sin embargo, **fgetc()** también devuelve **EOF** si ocurre un error. Puede usar **ferror()** para determinar con precisión lo que ha ocurrido.

foef()

Como se acaba de describir, **fgetc()** devuelve **EOF** cuando se ha encontrado el final del archivo. Sin embargo, la prueba del valor devuelto por **fgetc()** tal vez no sea la mejor manera de determinar cuando se ha llegado al final de un archivo. En primer lugar, el sistema de archivos de C puede operar en archivos de texto y binarios. Cuando se abre un archivo para entrada binaria, es posible leer un valor de entero que sea igual a **EOF**. Esto causaría que la rutina de entrada indique una condición de final de archivo aunque no se haya alcanzado el final físico del archivo. En segundo lugar, **fgetc()** devuelve **EOF** cuando falla *y* cuando alcanza el final del archivo. Si sólo se emplea el valor devuelto de **fgetc()**, es imposible saber qué ocurrió. Para resolver estos problemas, C incluye la función **feof()**, que determina cuando se ha encontrado el final del archivo. La función **feof()** se muestra a continuación:

int feof(FILE **aa*)

Devuelve true si se ha alcanzado el final del archivo; de otra manera, devuelve false. Por tanto, la siguiente instrucción lee un archivo binario hasta que se encuentre el final del archivo:

```
while(!feof(aa)) car = fgetc(aa);
```

Por supuesto, puede aplicar este método para archivos de texto, además de archivos binarios.

ferror()

La función **ferror()** determina si una operación con archivos ha producido un error. Aquí se muestra la función **ferror()**:

int ferror(FILE **aa*)

El parámetro *aa* especifica el archivo en cuestión. La función devuelve true si ha ocurrido un error durante la última operación con archivos; de otra manera, devuelve false.

Ejemplo

En el siguiente programa se ilustra el E/S de archivos de C. Se copia un archivo de texto. En el proceso, se eliminan tabuladores y se sustituye el número apropiado de espacios. Para usar el programa, especifique el nombre del archivo de entrada y el de salida, y el tamaño del tabulador en la línea de comandos.

```
// Demuestra el sistema de E/S de C.
//
// Este programa copia un archivo, sustituyendo tabuladores
// con espacios en el proceso. Utiliza el sistema de E/S de C
// para manejar la E/S de archivo.

#include <iostream>
#include <cstdio>
#include <cstdlib>

using namespace std;

int main(int argc, char *argv[])
{
  FILE *entrada, *salida;
  int tamtab;
  int cuentatab;
  char car;
  int estado_completo = 0;

  if(argc != 4) {
    cout << "Uso: detab <entrada> <salida> <tama\u00a4o del tabulador>\n";
    return 1;
  }

  if((entrada = fopen(argv[1], "rb"))==NULL) {
    cout << "No se puede abrir el archivo de entrada.\n";
    return 1;
  }

  if((salida = fopen(argv[2], "wb"))==NULL) {
    cout << "No se puede abrir el ar4chivo de salida.\n";
    fclose (entrada);
    return 1;
  }

  // Obtiene el tamaño del tabulador.
  tamtab = atoi(argv[3]);

  cuentatab = 0;

  do {
    // Lee un carácter del archivo de entrada.
    car = fgetc(entrada);

    if(ferror(entrada)) {
      cout << "Error al leer el archivo de entrada.\n";
      estado_completo = 1;
      break;
    }

    // Si se encuentra un tabulador, se da salida al número apropiado de espacios.
    if(car == '\t') {
```

```
      for(int i=cuentatab; i < tamtab; ++i) {
        // Escribe espacios en el archivo de salida.
        fputc(' ', salida);
      }
      cuentatab = 0;
    }
    else {
      // Escribe el carácter en el archivo de salida.
      fputc(car, salida);

      ++cuentatab;
      if(cuentatab == tamtab) cuentatab = 0;
      if(car == '\n' || car == '\r') cuentatab = 0;
    }

    if(ferror(salida)) {
      cout << "Error al escribir en el archivo de salida.\n";
      estado_completo = 1;
      break;
    }
  } while(!feof(entrada));

  fclose(entrada);
  fclose(salida);

  if(ferror(entrada) || ferror(salida)) {
    cout << "Error al cerrar un archivo.\n";
    estado_completo = 1;
  }

  return estado_completo;
}
```

Opciones

Puede leer y escribir bloques de datos usando el sistema de E/S de C con las funciones **fread()** y **fwrite()**. Aquí se muestran:

size_t fread(void **buf*, size_t *num_bytes*, size_t *cuenta*, FILE **aa*)
size_t fwrite(const void **buf*, size_t *num_bytes*, size_t *cuenta*, FILE **aa*)

Para **fread()**, *buf* es un apuntador a una región de la memoria que recibirá los datos del archivo. Para **fwrite()**, es un apuntador a la información que se escribirá en el archivo. El valor de *cuenta* determina cuántos elementos se leen o escriben, y cada elemento tiene *num_bytes* de longitud. El archivo sobre el que se actúa se especifica con *aa*. La función **fread()** devuelve el número de elementos leídos. Este valor puede ser menor que *cuenta* si se alcanza el final del archivo o si ocurre un error. La función **fwrite()** devuelve el número de elementos escritos. Este valor será igual a *cuenta* a menos que ocurra un error.

Hay versiones alternas de **fgetc()** y **fputc()** llamadas **getc()** y **putc()**. Funcionan igual que sus contrapartes, excepto que pueden implementarse como macros.

Puede realizar operaciones de acceso aleatorio empleando el sistema de E/S de C con **fseek()**. Aquí se muestra:

int fseek(FILE **aa*, long *despl*, int *origen*)

El archivo sobre el que se actúa está especificado por *aa*. El número de bytes de *origen* que se volverá la posición actual se pasa en *despl.* El valor de *origen* debe ser uno de los siguientes (definidos en **<ctsdio>**:

Origen	Nombre de macro
Inicio del archivo	SEEK_SET
Posición actual	SEEK_CUR
Final del archivo	SEEK_END

Por tanto, para buscar desde el principio del archivo, *origen* debe ser **SEEK_SET**. Para hacerlo desde la posición actual, use **SEEK_CUR** y para el final del archivo, use **SEEK_END**. La función **fseek()** devuelve cero cuando se tiene éxito y un valor diferente de cero si ocurre un error.

El sistema de E/S de C da soporte a varias funciones que permiten E/S formada. Probablemente ha visto antes algunas de ellas. Las dos que se encuentran con más frecuencia son **printf()**, que da salida a datos formados a la consola, y **scanf()**, que lee datos formados de la consola. También hay variaciones de éstas, llamadas **fprintf()** y **fscanf()**, que operan en un archivo, y **sprintf()** y **sscanf()**, que usan una cadena para entrada y salida. En el capítulo 6, en que se brindan soluciones para formación de datos, se presenta una breve revisión general de estas funciones.

Puede restablecer la posición actual del archivo al principio de éste al llamar a **rewind()**. Se muestra a continuación:

void rewind(FILE **aa*)

El archivo que se regresará a la posición inicial está especificado por *aa*.

Para limpiar un flujo usando el sistema de E/S de C, llame a **fflush()**, que se muestra a continuación:

int fflush(FILE **aa*)

Escribe el contenido de cualquier dato en búfer al archivo asociado con *aa*. Si llama a **fflush()**, y *aa* es nulo, todos los archivos abiertos para salida se limpian. La función **fflush()** devuelve cero si tiene éxito; de otra manera, devuelve **EOF**.

Puede cambiar el nombre de un archivo al llamar a **rename()**. Puede borrar un archivo al llamar a **remove()**. Estas funciones se describen en la siguiente solución.

Un punto final: aunque C++ da soporte a los sistemas de E/S de C y C++, debe seguir algunas directrices para evitar problemas. En primer lugar, una vez que se ha abierto un flujo empleando uno de los sistemas, sólo debe actuarse sobre las funciones definidas por ese sistema. En otras palabras, no debe mezclar E/S de C y C++ en *el mismo archivo*. En segundo lugar, en general, es mejor usar el sistema de E/S basado en clases de C++. Éste da soporte al sistema de E/S de C por razones de compatibilidad con los programas existentes de C. El sistema de E/S de C no está orientado a los programas de C++.

Cambie el nombre de un archivo y elimínelo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<cstdio>		int remove(const char **nombrear*) int rename(const char **nombreant*, const char **nombrenue*)

En la solución anterior se presentó una breve revisión general del E/S de archivos de C. Como se mencionó allí, C++ da soporte completo al sistema de E/S de C, de modo que suele ser mejor usar el sistema de E/S para C++. Sin embargo, hay dos funciones definidas por el sistema de E/S de C que ofrece soluciones similares a dos tareas comunes: cambiar el nombre de un archivo y borrarlo. Las funciones son **rename()** y **remove()**. Se declaran en **<cstdio>**, y en esta solución se muestra cómo usarlos.

Paso a paso

Para cambiar el nombre de un archivo se requiere un paso:

1. Llame a **rename()**, especificando el nombre actual del archivo y su nuevo nombre.

Para borrar un archivo se requiere un paso:

1. Llame a **remove()**, especificando el nombre del archivo que se eliminará.

Análisis

La función **rename()** cambia el nombre de un archivo. Aquí se muestra:

int rename(const char **nombreant*, const char **nombrenue*)

El nombre actual del archivo se pasa en *nombreant*. El nuevo se pasa en *nombrenue*. Devuelve cero si se tiene éxito y un valor diferente de cero, de otra manera. En general, el archivo debe cerrarse antes de tratar de cambiarle el nombre. Además, como regla general, no es posible cambiar el nombre de un archivo de sólo lectura. Más aún, no es posible dar a un archivo un nombre que ya esté siendo usado por otro archivo. En otras palabras, no puede crear una situación en que existan nombres duplicados de archivos en el mismo directorio.

La función **remove()** borra un archivo. Aquí se muestra:

int remove(const char **nombrear*)

Elimina del sistema el archivo cuyo nombre se especifique en *nombrear*. Devuelve cero si tiene éxito y un valor diferente de cero si no. El archivo debe cerrarse antes de hacer un intento de borrarlo. Como regla general, el archivo no debe ser de sólo lectura o encontrarse en otra situación en que se evite su eliminación.

Ejemplo

En el siguiente ejemplo se muestran **rename()** y **remove()** en acción. Se crea un archivo llamado **prueba.dat**. Luego, si el argumento de la línea de comandos es "cambiarnombre", se cambia el nombre de **prueba.dat** por **prueba2.dat**. Si el argumento de la línea de comandos es "borrar", se elimina **prueba2.dat**.

```
// Demuestra rename() y remove().

#include <iostream>
#include <cstdio>
#include <cstring>
#include <fstream>

using namespace std;

int main(int argc, char *argv[])
{
  int resultado;

  if(argc != 2) {
    printf("Uso: BorrarCambiarnombre <borrar/cambiarnombre>\n");
    exit(1);
  }

  ofstream archsalida("prueba.dat");

  if(!archsalida) {
    cout << "No se puede abrir el archivo prueba.dat.\n";
    return 1;
  }

  archsalida << "Escriba algunos datos en el archivo.";

  archsalida.close();

  if(!archsalida.good()) {
    cout << "Error al escribir en el archivo o cerrarlo.\n";
    return 0;
  }

  if(!strcmp("borrar", argv[1])) {
    resultado = remove("prueba2.dat");
    if(resultado) {
      cout << "No se puede eliminar el archivo.\n";
      return 1;
    }
  } else if(!strcmp("cambiarnombre", argv[1])) {
    resultado = rename("prueba.dat", "prueba2.dat");
    if(resultado) {
      cout << "No se puede cambiar el nombre del archivo.\n";
      return 1;
    }
  } else
```

```
    cout << "Argumento de l\u00a1nea de comandos no v\u00a0lido.\n";

  return 0;
}
```

Opciones

Todos los sistemas operativos proporcionan funciones de API de bajo nivel que eliminan y cambian el nombre de archivos. Pueden ofrecer un control muy fino sobre estas operaciones.
Por ejemplo, pueden permitirle especificar un descriptor de seguridad. Para un control detallado, tal vez quiera usar los primitivos del sistema operativo, en lugar de **remove()** o **rename()**.

En algunos entornos, puede usar **rename()** para cambiar el nombre de un directorio. También puede mover un archivo de un directorio a otro empleando **rename()**. Revise la documentación de su compilador para conocer los detalles.

CAPÍTULO 6

Formación de datos

Si está desplegando la hora y fecha, trabajando con valores monetarios o simplemente deseando limitar el número de dígitos decimales, la formación de datos es una parte importante de muchos programas. También es un aspecto de la programación que plantea muchas preguntas. Una razón es el tamaño y la complejidad del problema. Hay muchos tipos diferentes de datos, formatos y opciones. Otra razón es la riqueza de las capacidades de formación de C++. A menudo, hay más de una manera de producir un formato deseado. Por ejemplo, puede establecer varios atributos de formación al emplear funciones como **setf()**, **width()** o **precision()**, o con manipuladores de E/S, como **setw**, **fixed** o **showpos**. He aquí otro ejemplo: puede formar la fecha y hora al usar la biblioteca de ubicación de C++ o la función **strftime()** heredada de C. Francamente, elegir un método es a veces una decisión difícil, sobre todo cuando se incluye código heredado. Por supuesto, el beneficio de este soporte amplio y flexible para formación es que puede usar la mejor técnica para el trabajo a mano.

En este capítulo se examina el tema de la formación y se presentan soluciones que demuestran varias maneras de resolver diversas tareas de formación comunes. En el proceso, se describen aspectos de localización, incluido el uso de facetas. Aunque el énfasis principal está en las características de formación definidas por C++, también se incluye el método basado en C original.

He aquí las soluciones en este capítulo:

- Acceda a marcas de formato mediante funciones de miembro de flujo
- Despliegue valores numéricos en diversos formatos
- Establezca la precisión
- Establezca el ancho de campo y el carácter de relleno
- Justifique la salida
- Use los manipuladores de E/S para formar datos
- Forme valores numéricos para una configuración regional y de idioma
- Forme valores monetarios empleando la faceta **money_put**
- Use las facetas **moneypunct** y **numpunct**
- Forme la fecha y hora con la faceta **time_put**
- Forme datos en una cadena

- Forme la fecha y hora con **strftime()**
- Use **printf()** para formar datos

Nota importante antes de empezar. Como se explicó en el capítulo 5, el sistema de E/S de C++ está construido sobre clases genéricas que pueden operar sobre diferentes tipos de caracteres. Más aún, declara especializaciones de esas clases para **char** y **wchar_t**. Para mayor conveniencia, en este capítulo se usan exclusivamente las especializaciones de **char**. Por tanto, se usan los nombres de especialización de **char**, como **ios**, **ostream** e **istream** (en lugar de **basic_ios**, **basic_ostream**, **basic_istream**, etcétera). Sin embargo, la información también se aplica a flujos definidos en otros tipos de carácter.

Revisión general del formato

Hay varias maneras en que el formato de datos puede especificarse o afectarse. Puede ser:

- Usar funciones de miembro de flujo para establecer o limpiar una o más marcas de formato.
- Usar funciones de miembro de flujo para establecer el ancho de campo, la precisión y el carácter de relleno.
- Usar un manipulador de E/S dentro de una expresión de salida formada para establecer marcas de formato u otros atributos.
- Usar la funcionalidad definida por la biblioteca de localización de C++ para formar valores numéricos, monetarios y de fecha y hora.
- Usar la familia **printf()** de funciones, que se heredan del lenguaje C, para formar datos (excepto para fecha y hora).
- Usar **strftime()**, también heredado de C, para formar fecha y hora.

Todos éstos se demuestran con las soluciones de este capítulo, pero el eje principal está en los primeros cuatro porque representan el método moderno de formación que utiliza C++. Las funciones **printf()** y **strftime()**, que se heredan de C, se cubren también para dar una visión completa, pero casi todo el código nuevo debe usar las características de C++.

Aunque los detalles específicos de cada método de formación se describen en las soluciones, aquí se presenta una revisión general.

Las marcas de formato

Cada flujo está asociado con un conjunto de marcas de formato que controlan la manera en que se forma la información. Estas marcas están contenidas en una enumeración de máscara de bits llamada **fmtflags** que está definida por **ios_base**. (Consulte el capítulo 5 para conocer detalles sobre flujos y el sistema de E/S de C++, en general.) Aquí se muestran las marcas de formato:

boolalpha	dec	fixed	hex
internal	left	oct	right
scientific	showbase	showpoint	showpos
skipws	unitbuf	uppercase	

A continuación se presenta una breve descripción de cada marca. Varias se exploran de manera detallada en las soluciones.

Las marcas **left**, **right** e **internal** determinan la manera en que se justifican los datos dentro de un campo. Forman un grupo en que sólo uno debe establecerse en cualquier momento dado. Cuando está establecida la marca **left**, la salida se justifica a la izquierda. Cuando se establece **right**, la salida se justifica a la derecha. Cuando la marca **internal** se establece, el valor numérico se trata de manera especial para llenar un campo mediante la inserción de caracteres de relleno (que, como opción predeterminada, es un espacio) entre cualquier carácter de signo o de base. En muchas configuraciones regionales, la opción predeterminada es la justificación a la derecha.

Como opción predeterminada, se da salida a los valores numéricos en decimal, pero es posible seleccionar la base del número al usar las marcas **oct**, **hex** y **dec**. Estas marcas forman un grupo en que sólo uno debe establecerse en cualquier momento determinado. Cuando la marca **oct** se establece, la salida se despliega en octal. El establecimiento de la marca **hex** causa que la salida se despliegue en hexadecimal. Para regresar la salida a decimal, se establece la marca **dec**.

El establecimiento de **showbase** causa que se muestre la base de valores numéricos. En el caso de hexadecimales, un valor se antecederá con un 0x. Por ejemplo, 1F se desplegará como 0x1F. En el caso de octal, el valor se antecederá con un 0, como en 076. Los valores decimales no se ven afectados.

Como opción predeterminada, cuando se despliega la notación científica, la **e** está en minúsculas. Además, cuando se despliega un valor hexadecimal, la **x** está en minúsculas. Cuando **uppercase** está establecida, estos caracteres se despliegan en mayúsculas.

El establecimiento de **showpos** causa que un signo de más al principio se despliegue antes de los valores positivos.

El establecimiento de **showpoint** causa que se despliegue un punto decimal y ceros al principio en toda la salida de punto flotante (se necesiten o no).

El establecimiento de la marca **scientific** causa que se desplieguen los valores numéricos de punto flotante empleando notación científica. Cuando se establece **fixed**, los valores de punto flotante se despliegan usando notación de punto flotante. Estas marcas forman un grupo en que sólo debe establecerse una en un momento determinado. Cuando no se establece ninguna marca, el compilador elige un método apropiado.

Cuando se establece **unitbuf**, se limpia el búfer después de cada operación de inserción.

Cuando se establece **boolalpha**, puede darse entrada o salida a valores booleanos empleando las palabras clave **true** y **false**. De otra manera, se utilizan los dígitos 1 y 0.

La marca **skipws** se aplica a flujos de entrada. Cuando se establece, se descartan los caracteres de espacio en blanco al inicio (espacios, tabuladores y nuevas líneas) cuando se realiza entrada en un flujo. Cuando se limpia **skipws**, no se descartan.

También están definidos los valores **basefield**, **adjustfield** y **floatfield**. El **basefield** está definido como **oct | dec | hex**. Por tanto **basefield** le permite hacer referencia a los campos **oct**, **dec** y **hex** colectivamente. De manera similar, los campos **left**, **right** e **internal** están combinados en **adjustfield**. Por último, puede hacerse referencia a los campos **scientific** y **fixed** como **floatfield**. Como se demostrará en las soluciones, estos valores simplifican el establecimiento de una marca específica dentro de un grupo de marcas.

Las marcas de formato están definidas por **ios_base**, que es una clase de base para **basic_ios**. Como se explicó en el capítulo 5, el sistema de E/S de C++ crea especializaciones para flujos de tipo **char** y **wchar_t**. la especialización de **char** de **basic_ios** es **ios**. Por tanto, es común ver las marcas de formato a las que se hace referencia a través de **ios**, como en **ios::oct**. Éste es el método que se usará en este capítulo. (Aunque es perfectamente adecuado usar **ios_base::oct**, si lo prefiere.)

Los atributos de ancho de campo, precisión y carácter de relleno

Además de las marcas de formato que se acaban de describir, cada flujo de C++ está asociado con los tres atributos que afectan el formato. Son los atributos de ancho de campo, precisión y carácter de relleno. El ancho de campo especifica el número mínimo de caracteres que ocupará un elemento formado. Especifica el número mínimo de caracteres que ocupará un elemento formado. Como opción predeterminada, el ancho de campo es igual al número de caracteres en el elemento que se está desplegando, pero puede cambiar esto para que un elemento quede contenido dentro de un espacio mayor. Como opción predeterminada, el carácter usado para rellenar la salida es el espacio, pero puede cambiar esto. Por último, la precisión predeterminada de los valores de punto flotante es 6, pero esto, también, está bajo su control.

Funciones miembro de flujo relacionadas con formato

Cada flujo de C++ contiene su propio conjunto de marcas de formato y atributos ancho de campo, precisión y carácter de relleno. En el caso de cualquier flujo determinado, pueden establecerse las marcas de formato, limpiarse o interrogarse mediante el uso de las funciones **setf()**, **unsetf()** y **flags()**. Son miembros de **ios_base**. El ancho de campo se establece con **width()**, y la precisión con **precision()**. Ambas son miembros de **ios_base**. El carácter de relleno se establece con **fill()**, que es miembro de **ios**. Se describen con todo detalle en las soluciones.

Los manipuladores de E/S

Otra manera de establecer las marcas de formato y los atributos es mediante el uso de un manipulador. Un *manipulador* es una función (o, en algunos casos, un objeto) que se incluye en una expresión de E/S formada. Puede usarse para establecer o limpiar las marcas de formato o para afectar el flujo, de otra manera. C++ define varios manipuladores estándar. Se muestran a continuación:

boolalpha	dec	endl
ends	fixed	flush
hex	internal	left
nobooalpha	noshowbase	noshowpoint
noshowpos	noskipws	nounitbuf
nouppercase	oct	resetiosflags(fmtflags *f*)
right	scientific	setbase(int *base*)
setfill(int *car*)	setiosflags(fmtflags *f*)	setprecision(int *p*)
setw(int *w*)	showbase	showpoint
showpos	skipws	unitbuf
uppercase	ws	

Los manipuladores caen en dos categorías generales: con parámetros y sin ellos. Un manipulador con parámetros requiere un argumento cuando se usa. Un ejemplo de un manipulador con parámetros es **setw**. Establece el ancho de campo en el tamaño que se pasa. Un manipulador sin parámetros no toma un argumento. Por ejemplo, el manipulador **endl** no tiene un argumento. Casi ninguno de los manipuladores estándar toman argumentos.

Casi todos los manipuladores sin parámetros están definidos por el encabezado **<ios>**, que se incluye automáticamente en otros encabezados, como **<iostream>**. Tres están definidos por el encabezado **<ostream>**: **endl**, **ends** y **flush**. Los manipuladores con parámetros están definidos en

<iomanip>. Los manipuladores se describen de manera detallada en *Use manipuladores de E/S para formar datos.*

Forme datos utilizando la biblioteca de localización

Los datos de formato que rebasan las capacidades básicas proporcionadas por las marcas y los atributos de formato requieren el uso de una o más funciones y clases de biblioteca. En el caso de algunos tipos de formato, puede usar funciones heredadas de C (el lenguaje sobre el que se construyó C++). Su utilidad principal está en el mantenimiento de código heredado y se describen en las siguientes secciones. En el caso de nuevo código, por lo general usará las características de formato definidas por la *biblioteca de localización*. Esta biblioteca está definida en el encabezado **<locale>**, y proporciona soporte para formación de datos, como valores monetarios y fecha y hora, cuya representación es sensible a la cultura y el idioma.

La biblioteca de localización está basada en la clase **locale**, que define una configuración regional y de idioma. Esta configuración encapsula la información geográfica relacionada con un flujo. Es importante comprender que cada flujo tiene su propio objeto de **locale**. Por tanto, el establecimiento de la configuración regional y de idioma de un flujo afecta sólo a ese flujo. Esto difiere del lenguaje C, en que está disponible una configuración global (C++ aún le da soporte a ésta para proporcionar compatibilidad hacia atrás con C, pero las configuraciones regionales y de idioma basadas en flujo son mucho más flexibles.)

La clave para el uso de una instancia de **locale** para el manejo de la formación es la *faceta*. Una faceta es una instancia de una clase que hereda **locale::facet**. Cada faceta describe algún aspecto de la configuración regional y de idioma. Por ejemplo, la faceta que maneja formato monetario es **money_put**. La faceta que forma hora y fecha es **time_put**. Al usar una faceta, los datos pueden formarse como lo desee y también pueden adecuarse a la medida de una configuración específica. Esto le da mucha capacidad al subsistema de localización de C++. Una revisión general de las facetas se presentará en breve, y en las soluciones se brinda información específica acerca de las facetas que manejan valores numéricos y monetarios, además de fecha y hora.

La familia de funciones printf()

Debido a que C++ se construyó a partir de C, incluye todas las bibliotecas de funciones definidas por C. Esto significa que C++ da soporte a la familia de funciones **printf()**. Estas funciones son parte del sistema de E/S de C y proporcionan el mecanismo mediante el cual un programa de C forma datos. Aunque el uso de **printf()** no está recomendado para nuevo código de C++, es la función que usará cuando escriba programas de C. También se encuentra con frecuencia en código heredado. Por tanto, ningún libro de C++ estaría completo sin un análisis de sus características.

Hay varias funciones en **printf()**. Aquí se presentan las usadas en este capítulo:

printf()	Despliega salida formada en el dispositivo de salida estándar, que como opción predeterminada es la consola.
fprintf()	Escribe salida formada en un archivo.
sprintf()	Escribe salida formada en una cadena.

Todas requieren el encabezado **<cstdio>**, y todas funcionan de la misma manera básica. Es simplemente el destino de la salida lo que cambia. La operación de estas funciones se describe en *Use **printf()** para formar datos.*

NOTA *Las versiones de carácter amplio de la familia de funciones **printf()** también están disponibles. Por ejemplo, la versión de carácter ancho de **printf()** es **wprintf()**. Las versiones de carácter ancho usan el encabezado **<cwchar>**.*

La función strftime()

Otra función de formato heredada de C es **strftime()**. Forma información de fecha y hora. Aunque las facetas de C++, como **time_put**, proporcionan más flexibilidad, la función **strftime()** puede ser más fácil de usar en algunos casos. También suele encontrarse en código C heredado. Se describe en *Forme fecha y hora usando **strftime()**.*

Revisión general de las facetas

Las facetas son los medios para la formación de los datos en C++. Son parte de la biblioteca de localización, que requiere el encabezado **<locale>**. Tal vez lo más importante que debe comprender acerca de las facetas es que resultan más fáciles de usar de lo que parece a primera vista. No se intimide con su más bien compleja sintaxis de plantilla. Una vez que comprende el proceso general, es fácil crear cualquier tipo de formato localizado que desee. Debido a que son varias las soluciones en que se usan las facetas, tiene sentido describir el procedimiento general en un lugar, y describir los detalles específicos en las soluciones individuales.

Todas las facetas son clases que se derivan de **locale::facet**. Hay varias facetas integradas, como **money_put**, **time_get** y **num_put**, que están declaradas en **<locale>**. Estas clases se usan para formar datos para salida o leer datos formados de la entrada. En este capítulo sólo se trata la formación de datos para salida, de modo que aquí no se usan las facetas de entrada. Más aún, en este capítulo sólo se utilizan las facetas que forman valores numéricos y monetarios, además de fecha y hora. La biblioteca de localización define otras facetas que manejan otras necesidades sensibles a la configuración de región e idioma.

Conceptualmente, el uso de una faceta es fácil: se obtiene una faceta al llamar a **use_facet()** y luego se llaman a funciones de esa faceta para formar datos u obtener información de localización. Sin embargo, en la práctica, el proceso suele ser un poco más complejo. He aquí un esquema general de estos pasos:

1. Construya un objeto de **locale**.
2. Establezca la configuración regional y de idioma deseada al llamar a **imbue()** en el flujo que estará recibiendo la salida formada. Pase **imbue()** al objeto de **locale** del paso 1.
3. Obtenga una faceta al llamar a **use_facet()**, especificando el nombre de la faceta. Se trata de una función global definida por **<locale>**.
4. Para formar valores numéricos y monetarios, o la fecha y hora, o para obtener información acerca de un formato, utilice la función definida por la faceta obtenida en el paso 3.

Revisemos más de cerca todos los pasos.

La clase **locale** define varios constructores. Aquí se muestra el usado en este capítulo:

explicit locale(const char **nombre_loc*)

El nombre de la configuración regional y de idioma se pasa mediante *nombre_loc*. Debe ser un nombre válido. Si no lo es, se lanza un **runtime_error**. Lo que constituye un nombre válido depende de la implementación. En este libro se usan cadenas de configuración regional y de idioma que

son compatibles con Visual C++ de Microsoft. Necesitará revisar la documentación de su compilador para conocer las cadenas a las que da soporte.

Para establecer una configuración regional y de idioma del flujo, llame a **imbue()**. Está definida por **ios_base** y se encuentra disponible en todos los objetos de flujo. El proceso para el establecimiento de esta configuración se describe de manera detallada en *Obtenga o establezca una configuración regional y de idioma de flujo* en el capítulo 5. Es conveniente mostrar **imbue()** aquí una vez más:

locale imbue(const locale &*locnue*)

Se devuelven la configuración regional y de idioma del flujo que invoca, y la configuración anterior.

Para obtener una faceta, llame a **use_facet()**. Es una función global y se muestra aquí:

template <class Facet> const Facet &use_facet(const locale &*loc*)

Aquí, **Facet** debe ser una faceta válida. Especifica la faceta que se obtendrá, que normalmente será definida por **<locale>**. (Es posible crear facetas personalizadas, pero rara vez necesitará hacerlo.) La configuración regional y de idioma para la que se obtendrá la faceta se pasa en *loc*. La función **use_facet()** devuelve una referencia a la faceta especificada por **Facet**. Si ésta no existe, se lanza **bad_cast**. (Si es necesario, puede determinar si una faceta existe al llamar a **has_facet()**, que también es una función global definida por **<locale>**.)

Hay varias facetas predefinidas. Las usadas en este libro son:

num_put	Forma valores numéricos.
money_put	Forma valores monetarios.
time_put	Forma fecha y hora.
numpunct	Obtiene signos de puntuación y reglas relacionadas con los formatos numéricos.
moneypunct	Obtiene signos de puntuación y reglas relacionadas con los formatos monetarios.

Las soluciones muestran sus declaraciones, con excepción de todas las clases de plantilla que toman el tipo de carácter como argumento de tipo. (Algunas también tienen otro tipo de argumento.) Las facetas **num_put**, **money_put** y **time_put** forman números, dinero y hora y fecha, respectivamente. Definen la función **put()**, que forma el valor que se pasa de acuerdo con las reglas encapsuladas por la faceta. (Cada una de las funciones de **put()** se describe en su propia solución.) La faceta **numpuct** encapsula información acerca de los signos de puntuación y las reglas que determinan el formato de los datos numéricos. La faceta **moneypunct** encapsula los signos de puntuación y las reglas que rigen el formato de valores monetarios.

Para obtener una faceta, utilizará **use_facet()**, especificando el nombre de la faceta como parámetro de tipo. Por ejemplo, ésta obtiene una faceta **money_put** asociada con la configuración regional y de idioma usada por **cout**:

```
const money_put<char> &mp = use_facet<money_put<char> >(cout.getloc());
```

Observe que la versión **char** de **money_put** es obligatoria porque **cout** es un flujo de **char**. Una vez que tiene una faceta, puede usarla para formación al llamar a funciones en ella. En esta solución se describe el proceso de manera detallada.

He aquí un tema muy importante: cuando se usa un flujo de C++, se da salida automática a los números al usar la faceta **num_put**. Por tanto, no necesita obtener manualmente esta faceta al desplegar valores numéricos de una manera que sea específica de la configuración regional y de idioma. Simplemente establezca la configuración del flujo al usar **imbue()** y el valor se formará automáticamente para esa configuración.

NOTA *También puede establecer globalmente la configuración regional y de idioma, empleando la función heredada de C* ***setlocale()****. Sin embargo, este método no se recomienda para nuevo código. El sistema de configuración regional y de idioma de la faceta usada por C++ ofrece un método mejor y más flexible.*

Acceda a las marcas de formato mediante las funciones de miembro de flujo

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>	ios_base	fmtflags setf(fmtflags *marcas*) void unsetf(fmtflags *marcas*) fmtflags flags()

Para cualquier flujo determinado, puede cambiar la manera en que se forman los datos al cambiar una o más marcas de formato. Por ejemplo, si establece la marca **showpos**, entonces los valores numéricos positivos se despliegan con un signo **+** al principio. Hay dos maneras en que pueden establecerse las marcas de formato. En primer lugar, puede usar funciones que están definidas por todas las clases de flujo, como **setf()**. En segundo lugar, puede usar un manipulador de E/S. En esta solución se muestra cómo usar las funciones miembro del flujo. Los manipuladores se describen en una solución posterior.

Paso a paso

Para usar las funciones miembro de flujo para establecer, limpiar u obtener las marcas de formato, se requieren estos pasos:

1. Para establecer una o más marcas en un flujo, llame a **setf()**.
2. Para limpiar una o más marcas en un flujo, llame a **unsetf()**.
3. Para obtener la configuración de marca de formato actual, llame a **flags()**.

Análisis

Para cualquier flujo determinado, puede establecer una marca de formato al llamar a la función **setf()**, que se declara con **ios_base**. Por tanto, **setf()** es un miembro de todas las clases de flujo. Aquí se muestra:

fmtflags setf(fmtflags *marcas*)

Esta función devuelve la configuración anterior de las marcas de formato y habilita esas marcas especificadas por *marcas*. Por ejemplo,

```
miflujo.setf(ios::showpos);
```

habilita la marca **showpos** para el flujo llamado **miflujo**.

El complemento de **setf()** es **unsetf()**. También se declara con **ios_base**. Limpia una o más marcas de formato. Su forma general es:

void unsetf(fmtflags *marcas*)

Se limpian las marcas especificadas por *marcas*. Todas las otras marcas quedan sin afectación. Por tanto, para deshabilitar la marca **boolalpha** para **miflujo**, utilizaría esta instrucción:

```
miflujo.unsetf(ios::boolalpha);
```

Puede establecer o limpiar más de una marca en una sola llamada a **setf()** o **unsetf()** al emplear juntas con el operador lógico OR dos o más marcas. Por ejemplo, esto habilita las marcas **showpos** y **boolalpha:**

```
miflujo.setf(ios::showpos | ios::boolalpha)
```

Lo siguiente deshabilita las marcas **uppercase** y **boolalpha**:

```
miflujo.unsetf(ios::uppercase | ios::boolalpha);
```

Puede obtener la configuración de la marca de formato actual al usar **flags()**. Aquí se muestra:

fmtflags flags() const

Devuelve la máscara de bits de marca de formato. También se declara con **ios_base**:

Es importante comprender que cada instancia de flujo tiene su propio conjunto de marcas de formato. Por tanto, el cambio de la configuración de marca para un flujo afecta sólo a ese flujo. Las marcas de formato de cualquier otro flujo quedan sin cambio.

Ejemplo

En el siguiente ejemplo se muestra cómo establecer y limpiar marcas de formato. Primero establece la marca **boolalpha** en **cout** y luego despliega un valor **bool**. Luego limpia la marca **boolalpha** y vuelve a desplegar el valor. Observe la diferencia en la salida.

```
// Demuestra las funciones setf() y unsetf().

#include <iostream>

using namespace std;

int main()
{

  // Establece la marca boolalpha en cout.
  cout.setf(ios::boolalpha);

  cout << "El valor true cuando se establece la marca boolapha: "
       << true << endl;;

  // Ahora, limpia la marca boolalpha.
  cout.unsetf(ios::boolalpha);

  cout << "El valor true cuando se limpia la marca boolapha: "
       << true << endl;;

  return 0;
}
```

Aquí se muestra la salida:

```
El valor true cuando se establece la marca boolapha: true
El valor true cuando se limpia la marca boolapha: 1
```

Ejemplo adicional: despliegue la configuración de la marca de formato

Cuando se depuran problemas de formato, en ocasiones resulta útil ver cómo están establecidas todas las marcas. De acuerdo con la experiencia del autor, algunos compiladores se comportan de maneras inesperadas debido a la interacción entre marcas aparentemente sin relación. Además, puede haber diferencias entre compiladores cuando dos marcas entran en conflicto. Por ejemplo, si las marcas **oct** y **dec** están establecidas, ¿cuál formato se usa? Diferentes compiladores podrían resolver esta situación de manera distinta. (Por supuesto, una buena práctica de programación determina que sólo una de las marcas **oct**, **dec** o **hex** se establezca en cualquier momento.) El hecho de poder ver las configuraciones de marcas reales puede ayudar a explicar resultados que de otra manera parecerían poco usuales. Con este fin, en el siguiente programa se crea una función llamada **mostrarmarcas()**, que toma un flujo como argumento y despliega la configuración actual de las marcas de formato de ese flujo:

```
// Este programa crea una función llamada mostrarmarcas()
// que despliega la configuración de marca de formato
// asociada con un flujo determinado.

#include <iostream>

using namespace std;

void mostrarmarcas(ios &flujo) ;

int main()
{
  // Muestra la condición predeterminada de marcas de formato.
  cout << "Configuraci\u00a2n predeterminada para cout:\n";
  mostrarmarcas(cout);

  // Establece las marcas right, showpoint y fixed.
  cout.setf(ios::right | ios::showpoint | ios::fixed);

  // Muestra las marcas después de llamar a setf().
  cout << "Marcas tras establecer right, showpoint y fixed:\n";
  mostrarmarcas(cout);

  return 0;
}

// Esta función despliega el estatus de las marcas de formato
// para el flujo especificado.
void mostrarmarcas(ios &flujo)
{
  ios::fmtflags f;

  // Obtiene la configuración de marcas actual.
  f = flujo.flags();

  if(f & ios::boolalpha) cout << "boolalpha:\thabilitada\n";
  else cout << "boolalpha:\tdeshabilitada\n";

  if(f & ios::dec) cout << "dec:\t\thabilitada\n";
  else cout << "dec:\t\tdeshabilitada\n";
```

```
  if(f & ios::hex) cout << "hex:\t\thabilitada\n";
  else cout << "hex:\t\tdeshabilitada\n";

  if(f & ios::oct) cout << "oct:\t\thabilitada\n";
  else cout << "oct:\t\tdeshabilitada\n";

  if(f & ios::fixed) cout << "fixed:\t\thabilitada\n";
  else cout << "fixed:\t\tdeshabilitada\n";

  if(f & ios::scientific) cout << "scientific:\thabilitada\n";
  else cout << "scientific:\tdeshabilitada\n";

  if(f & ios::right) cout << "right:\t\thabilitada\n";
  else cout << "right:\t\tdeshabilitada\n";

  if(f & ios::left) cout << "left:\thabilitada\n";
  else cout << "left:\t\tdeshabilitada\n";

  if(f & ios::internal) cout << "internal:\thabilitada\n";
  else cout << "internal:\tdeshabilitada\n";

  if(f & ios::showbase) cout << "showbase:\thabilitada\n";
  else cout << "showbase:\tdeshabilitada\n";

  if(f & ios::showpoint) cout << "showpoint:\thabilitada\n";
  else cout << "showpoint:\tdeshabilitada\n";

  if(f & ios::showpos) cout << "showpos:\thabilitada\n";
  else cout << "showpos:\tdeshabilitada\n";

  if(f & ios::uppercase) cout << "uppercase:\thabilitada\n";
  else cout << "uppercase:\tdeshabilitada\n";

  if(f & ios::unitbuf) cout << "unitbuf:\thabilitada\n";
  else cout << "unitbuf:\tdeshabilitada\n";

  if(f & ios::skipws) cout << "skipws:\t\thabilitada\n";
  else cout << "skipws:\t\tdeshabilitada\n";

  cout << " \n";
}
```

Aquí se muestra la salida. (Ésta se generó con Visual C++. Su compilador puede mostrar configuraciones predeterminadas diferentes.)

```
Configuración predeterminada para cout:
boolalpha:      deshabilitada
dec:            habilitada
hex:            deshabilitada
oct:            deshabilitada
fixed:          deshabilitada
scientific:     deshabilitada
right:          deshabilitada
left:           deshabilitada
```

```
internal:       deshabilitada
showbase:       deshabilitada
showpoint:      deshabilitada
showpos:        deshabilitada
uppercase:      deshabilitada
unitbuf:        deshabilitada
skipws:         habilitada

Marcas tras establecer right, showpoint y fixed:
boolalpha:      deshabilitada
dec:            habilitada
hex:            deshabilitada
oct:            deshabilitada
fixed:          habilitada
scientific:     deshabilitada
right:          habilitada
left:           deshabilitada
internal:       deshabilitada
showbase:       deshabilitada
showpoint:      habilitada
showpos:        deshabilitada
uppercase:      deshabilitada
unitbuf:        deshabilitada
skipws:         habilitada
```

Opciones

Hay una versión sobrecargada de **setf()** que toma esta forma general:

fmtflags setf(fmtflags *marcas1*, fmtflags *marcas2*)

En esta versión, sólo las marcas especificadas por *marcas2* se ven afectadas. Primero se limpian y luego se establecen de acuerdo con las marcas especificadas por *marcas1*. Tome nota de que, aunque *marcas1* contiene otras marcas, sólo las especificadas por *marcas2* se verán afectadas. Se devuelve la configuración de marcas anterior. Tal vez el uso más común de la forma de dos parámetros de **setf()** sea cuando establece las marcas de formato de base de número, justificación y punto flotante. Consulte las siguientes soluciones para conocer más detalles.

Puede establecer todas las marcas de formato al usar esta versión sobrecargada de **flags()**:

ftmflags flags(ftmflags *marcas*)

Esta versión asigna el valor pasado en *marcas* a toda la máscara de bits de marcas de formato. Se devuelve la máscara de bits anterior.

Las marcas de formato pueden establecerse mediante varios manipuladores. Por ejemplo, el manipulador **noboolalpha** limpia la marca **boolalpha**. También puede establecer o limpiar una o más marcas empleando los manipuladores **setiosflags** y **resetiosflags**. Consulte *Use manipuladores de E/S para formar datos*.

Despliegue valores numéricos en diversos formatos

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>	ios_base	fmtflags setf(fmtflags *marcas*) void unsetf(fmtflags *marcas*) oct hex dec showbase showpos fixed scientific basefield floatfield

Mediante el uso de marcas de formato, se controlan varios aspectos del formato numérico. Por ejemplo, puede dar salida a enteros en hexadecimal u octal o desplegar valores de punto flotante en notación fija o científica. En esta solución se demuestran estas marcas que afectan el formato de números.

Paso a paso

El uso de marcas de formato para cambiar el formato de datos numéricos requiere estos pasos:

1. Para formar un entero en decimal, limpie las marcas especificadas por **basefield** y luego establezca la marca **dec**. Por lo general, el formato decimal es la opción predeterminada para un flujo de salida.
2. Para formar un entero en hexadecimal, limpie las marcas especificadas por **basefield** y luego establezca la marca **hex**.
3. Para formar un entero en octal, limpie las marcas especificadas por **basefield** y luego establezca la marca **oct**.
4. Para mostrar la base de un valor octal o hexadecimal, establezca la marca **showbase**.
5. Para formar un valor de punto flotante en notación fija, limpie las marcas especificadas por **basefield** y luego establezca la marca **fixed**.
6. Para formar un valor de punto flotante en notación científica, limpie las marcas especificadas por **basefield** y luego establezca la marca **scientific**.
7. Para hacer que un signo + se despliegue antes de los valores positivos, establezca la marca **showpos**.
8. Para asegurar que el punto decimal esté siempre incluido en un valor de punto flotante, establezca la marca **showpoint**.

9. Para que se muestren en mayúsculas las letras en valores numéricos (dígitos hexadecimales mayores que 0, la **e** en notación científica, y la **x** en el indicador de la base hexadecimal), establezca la marca **uppercase**.

Análisis

Las marcas de formato se establecen o limpian con las funciones **setf()** y **unsetf()**, que se describen de manera detallada en la solución anterior.

En general, puede desplegar valores enteros en decimal (la opción predeterminada), hexadecimal u octal. Esto se controla con el establecimiento de las marcas **dec**, **hex** y **oct**, respectivamente. Para establecer la base del número, debe habilitar la marca deseada y deshabilitar las otras dos. Por ejemplo, para dar salida a enteros en octal, debe habilitar **oct** y deshabilitar **dec** y **hex**. Colectivamente, se hace referencia a las marcas **oct**, **hex** y **dec** como **basefield**.

La manera más fácil de habilitar una marca y asegurarse de que las otras dos están deshabilitadas consiste en usar la forma de dos argumentos **setf()**. Como se explicó en la solución anterior, tiene esta forma general:

fmtflags setf(fmtflags *marcas1*, fmtflags *marcas2*)

En esta versión, sólo las marcas especificadas por *marcas2* se ven afectadas. Primero se limpian y luego se establecen de acuerdo con las marcas especificadas por *marcas1*. Por tanto, para establecer la base de un número, pasará **basefield** a *marcas2* (que hace que se limpien las marcas **oct**, **hex** y **dec**) y pasará la marca deseada de base de número en *marcas1*. Por ejemplo, lo siguiente establece la base del número de **cout** en hexadecimal:

```
cout.setf(ios::hex, ios::basefield);
```

Después de esta llamada, se establecerá la marca **hex** y se limpiarán las marcas **dec** y **oct**. Esto significa que toda la salida de enteros a **cout** se desplegará en hexadecimal.

Cuando se despliegan enteros, hará que la base se muestre al establecer la marca **showbase**. Cuando se establece, los valores desplegados en octal empiezan con un cero a la izquierda. Los valores desplegados en hexadecimal empiezan con un 0x. Los valores decimales no se ven afectados.

Como opción predeterminada, los valores de punto flotante se forman en formato de punto fijo o en notación científica, lo que sea más corto. Puede especificar la representación de punto fijo al establecer la marca **fixed**. Puede especificar notación científica al establecer la marca **scientific**. En cualquier caso, la otra marca debe deshabilitarse. La manera más fácil de hacer esto es usar la forma de dos argumentos de **setf()**, especificando que se deshabilitan las marcas de **floatfield**. Recuerde que **floatfield** combina las marcas **fixed** y **scientific**.

Para que un signo + anteceda a los valores positivos, establezca la marca **showpos**. En general, **showpos** sólo afecta los valores de punto flotante y los enteros desplegados en decimal. Los desplegados en octal o hexadecimal no se verán afectados.

Para que se despliegue un punto decimal, aunque no haya dígitos fraccionales, establezca la marca **showpoint**.

Como opción predeterminada, se despliegan en minúsculas las letras en valores numéricos, que incluyen los dígitos hexadecimales de la a a la f, la **e** en notación científica y la **x** en el indicador de base hexadecimal. Para cambiar a mayúsculas, especifique la marca **uppercase**.

Ejemplo

En el siguiente ejemplo se muestran en acción las marcas de formato numérico:

```
// Demuestra las marcas de formato numérico.
//
// En este ejemplo se muestra cout, pero debe
// sustituirse cualquier flujo de salida.

#include <iostream>

using namespace std;

int main()
{
  int x = 100;
  double f = 98.6;
  double f2 = 123456.0;
  double f3 = 1234567.0;

  cout.setf(ios::hex, ios::basefield);
  cout << "x en hexadecimal: " << x << endl;

  cout.setf(ios::oct, ios::basefield);
  cout << "x en octal: " << x << endl;

  cout.setf(ios::dec, ios::basefield);
  cout << "x en decimal: " << x << "\n\n";

  cout << "f, f2 y f3 en el formato predeterminado de punto flotante:\n";
  cout << "f: " << f << "  f2: " << f2 << "  f3: " << f3 << endl;

  cout.setf(ios::scientific, ios::floatfield);
  cout << "Tras establecer la marca scientific:\n";
  cout << "f: " << f << "  f2: " << f2 << "  f3: " << f3 << endl;

  cout.setf(ios::fixed, ios::floatfield);
  cout << "Tras establecer la marca fixed:\n";
  cout << "f: " << f << "  f2: " << f2 << "  f3: " << f3 << "\n\n";

  // Vuelve al formato de punto flotante predeterminado.
  cout << "Regresando al formato predeterminado de punto flotante.\n";
  cout.unsetf(ios::fixed);

  cout << "f2 en formato predeterminado: " << f2 << "\n\n";

  // Establece la marca showpoint.
  cout << "Estableciendo la marca showpoint.\n";
  cout.setf(ios::showpoint);
  cout << "f2 con showpoint establecido: " << f2 << "\n\n";

  cout << "Limpiando la marca showpoint.\n\n";
  cout.unsetf(ios::showpoint);

  // Establece la marca showpos.
  cout.setf(ios::showpos);
  cout << "Estableciendo la marca showpos.\n";
  cout << "x en decimal tras establecer showpos: " << x << endl;
```

```
  cout << "f en notaci\u00a2n predeterminada tras establecer showpos: " << f <<
"\n\n";

  // Establece la marca uppercase.
  cout << "Estableciendo la marca uppercase.\n";
  cout.setf(ios::uppercase);
  cout << "f3 con la marca uppercase establecida: " << f3 << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
x en hexadecimal: 64
x en octal: 144
x en decimal: 100

f, f2 y f3 en el formato predeterminado de punto flotante:
f: 98.6  f2: 123456  f3: 1.23457e+006
Tras establecer la marca scientific:
f: 9.860000e+001  f2: 1.234560e+005  f3: 1.234567e+006
Tras establecer la marca fixed:
f: 98.600000  f2: 123456.000000  f3: 1234567.000000

Regresando al formato predeterminado de punto flotante.
f2 en formato predeterminado: 123456

Estableciendo la marca showpoint.
f2 con showpoint establecido: 123456.

Limpiando la marca showpoint.

Estableciendo la marca showpos.
x en decimal tras establecer showpos: +100
f en notación predeterminada tras establecer showpos: +98.6

Estableciendo la marca uppercase.
f3 con la marca uppercase establecida: +1.23457E+006
```

Opciones

Las marcas de formato numérico pueden establecerse mediante manipuladores. Por ejemplo, la marca **showpoint** puede establecerse con el manipulador **showpoint** y limpiarse con el **noshowpoint**. Consulte *Use manipuladores de E/S para formar datos* para conocer más detalles.

Para cualquier flujo determinado, la precisión predeterminada es de 6 dígitos, pero puede cambiar esto al llamar a la función **precision()**. Consulte *Establezca la precisión* para conocer más detalles. También puede especificar un ancho de campo en que el valor se despliega al llamar a **width()** y el carácter de relleno utilizado para rellenar campos que son más largos que la salida al llamar a **fill()**. Se describen en *Establezca el ancho de campo y el carácter de relleno*.

Establezca la precisión

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>	ios_base	streamsize precision(streamsize *prec*)

Cada flujo tiene una configuración de precisión asociada que determina cuántos dígitos se despliegan cuando se forma un valor de punto flotante. La precisión predeterminada es 6. Puede cambiar esto al llamar a **precision()**. Como se explica en el análisis que sigue, el significado exacto de la precisión difiere de acuerdo con el formato de punto flotante que se use.

Paso a paso

Para establecer la precisión se necesitan estos pasos:

1. Establezca la precisión al llamar a **precision()** en el flujo.
2. En algunos casos, tal vez necesite ajustar el formato de punto flotante al establecer la marca **fixed** o **scientific** para lograr los resultados deseados.

Análisis

Cada flujo tiene su propio atributo de precisión. La precisión se establece al llamar a **precision()** en el flujo. Esta función es un miembro de **ios_base** y se hereda de todas las clases de flujo. Aquí se muestra una de sus formas:

streamsize precision(streamsize *prec*)

La precisión del flujo que invoca se establece con *prec*. Se devuelve la precisión anterior. La precisión predeterminada de un flujo es 6. El tipo **streamsize** está definido como alguna forma de entero que puede contener el número más largo de bytes que puede transferirse en cualquier operación de E/S.

El efecto de la precisión se basa en el formato de punto flotante que se está usando. En el caso del formato predeterminado, la precisión determina el número de dígitos significativos desplegados. En la notación de punto fijo o científica, la precisión determina el número de dígitos desplegados a la derecha del punto decimal. (La notación científica se utiliza cuando se establece la marca **scientific** y se ha limpiado la marca **fixed**. La notación de punto fijo se usa cuando se ha limpiado la marca **scientific** y se establece la marca **fixed**.)

El establecimiento de la precisión responde una de las preguntas tipo "¿cómo hacer?" más comunes: "¿Cómo despliego dos números decimales?" Esto se logra fácilmente al establecer la marca **fixed** y luego definiendo la precisión en 2. Después de hacer esto, se desplegarán dos números decimales en todos los casos, aunque no haya dígitos decimales significativos. De manera más general, si necesita especificar un número fijo de dígitos decimales, entonces establezca la marca **fixed** y especifique el número de dígitos en una llamada a **precision()**.

Ejemplo

En el ejemplo siguiente se muestran los efectos del establecimiento de la precisión:

```
// Demuestra el establecimiento de la precisión.

#include <iostream>

using namespace std;

int main()
{
  double f = 123456.123456789;

  cout << "Usando el formato de n\u00a3mero predeterminado.\n";
  cout << "f con precisi\u00a2n predeterminada: " << f << "\n\n";

  cout << "Estableciendo la precisi\u00a2n de 9.\n";
  cout.precision(9);
  cout << "f con precisi\u00a2n de 9: " << f << "\n\n";

  cout << "Cambiando a formato de punto fijo.\n";
  cout.setf(ios::fixed, ios::floatfield);

  cout << "f con precisi\u00a2n de 9 en punto fijo: " << f << "\n\n";

  // Ahora, despliega dos lugares decimales.
  cout << "Despliega dos lugares decimales en todos los casos: ";
  cout.precision(2);
  cout << 12.456 << " " << 10.0 << " " << 19.1 << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Usando el formato de número predeterminado.
f con precisión predeterminada: 123456

Estableciendo la precisión de 9.
f con precisión de 9: 123456.123

Cambiando a formato de punto fijo.
f con precisión de  9 en punto fijo: 123456.123456789

Despliega dos lugares decimales en todos los casos: 12.46 10.00 19.10
```

Opciones

Hay una segunda forma de **precision()**, que se muestra aquí:

streamsize precision() const

Esta forma devuelve la posición actual, pero no la cambia.

Otra manera de establecer la precisión de un flujo consiste en usar el manipulador de E/S **setprecision**. Se describe en *Use los manipuladores de E/S para formar datos*.

Establezca el ancho de campo y el carácter de relleno

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>	ios_base	streamsize width(streamsize *a*)
<ios>	ios	char fill(char *car*)

En esta solución se muestra cómo especificar un ancho de campo y un carácter de relleno. Como opción predeterminada, cuando se da salida a un valor, sólo ocupa el espacio adecuado para el número de caracteres que se requiere para desplegarlo. Esto suele ser exactamente lo que se quiere. Sin embargo, en ocasiones querrá que el valor llene un cierto ancho de campo, como cuando desea que se alineen columnas de datos. Aunque hay varias maneras de lograr esa salida, por mucho la más fácil consiste en especificar un ancho de campo. Una vez hecho esto, cada elemento se rellenará automáticamente para que ocupe todo el ancho de campo. El carácter de relleno predeterminado es un espacio, y esto suele ser lo que se quiere, pero puede cambiarlo.

Paso a paso

Para especificar el ancho de campo y el carácter de relleno se requieren estos pasos:

1. Para especificar un ancho de campo, llame a **width()** en el flujo.
2. Para especificar un carácter de relleno, llame a **fill()** en el flujo.

Análisis

Puede especificar un ancho de campo mínimo empleando la función **width()**. Tiene dos formas. Aquí se muestra la usada en esta solución:

streamsize width(streamsize *a*)

Aquí, *a* se vuelve el ancho de campo y se devuelve el ancho de campo anterior. Como regla general, el ancho de campo debe establecerse de inmediato antes de dar salida al elemento al que desea aplicar el ancho. Después de que se da salida a ese elemento, el ancho de campo se regresa a su opción predeterminada. (Se han visto implementaciones en que un solo establecimiento del ancho de campo se aplica a toda la salida posterior, pero es un comportamiento no estándar.) El tipo **streamsize** es un **typedef** para alguna forma de entero.

Después de que establezca un ancho de campo mínimo, cuando un valor usa menos del ancho especificado, el campo se rellenará con el carácter de relleno actual (un espacio, como opción predeterminada) para lograr el ancho deseado. Si el tamaño del valor excede el ancho de campo mínimo, entonces se rebasará el campo. Los valores no se truncan.

En la configuración regional y de idioma predeterminada, la salida se alinea a la derecha. Esto significa que si un campo necesita rellenarse para alcanzar un ancho especificado, entonces los caracteres de relleno se agregarán a la izquierda de los datos. Cuando la salida está alineada a la izquierda, los caracteres de relleno se agregarán a la derecha de los datos. Cuando está establecida la marca **internal**, se agrega el relleno en el interior de algunos tipos de formatos numéricos. Por ejemplo, si la marca **showpos** está establecida, entonces el relleno tiene lugar entre el signo + y los dígitos. Consulte *Justifique salida* para conocer más detalles.

Cuando se necesita rellenar un campo, se hace con el carácter de relleno, que es un espacio, como opción predeterminada. Puede especificar un carácter diferente empleando la función **fill()**. Tiene dos formas. Ésta es la usada aquí:

char fill(char *car*)

Después de una llamada a **fill()**, *car* se vuelve el nuevo carácter de relleno y se devuelve el anterior.

Ejemplo

En el siguiente ejemplo se demuestra el establecimiento del ancho de campo y el carácter de relleno. Hay dos cosas importantes que se deben observar en este programa. En primer lugar, una llamada a **width()** afecta sólo a la salida del siguiente elemento. En segundo lugar, el carácter de relleno se agrega entre el signo **+** y los dígitos cuando se despliegan los datos numéricos, en caso de que estén establecidas las marcas **internal** y **showpos**.

```
// Demuestra width() y fill().

#include <iostream>

using namespace std;

int main()
{
  // Usa el ancho predeterminado.
  cout << "Hola" << endl;

  // Establece el ancho en 10.
  cout.width(10);
  cout << "Hola" << endl;

  // Observe cómo el ancho regresa a la opción predeterminada
  // después de que se da salida a un elemento.
  cout << "Hola" << endl;

  // Ahora establece el ancho y el carácter de relleno.
  cout.width(10);
  cout.fill('*');
  cout << "Hola" << endl;

  // Observe que el carácter de relleno sigue establecido.
  cout.width(12);
  cout << 123.45 << endl;

  // Ahora, rellena el ancho de campo con espacios
```

```
  // y establece las marcas internal y showpos.
  cout.width(12);
  cout.fill(' ');
  cout.setf(ios::showpos | ios::internal);
  cout << 765.34 << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Hola
      Hola
Hola
******Hola
******123.45
+     765.34
```

Ejemplo adicional: alinee columnas de números

Uno de los usos más comunes de un ancho de campo mínimo consiste en crear tablas en que las columnas de números se alinean una sobre otra. Para ello, simplemente especifique un ancho de campo que sea por lo menos del tamaño del número máximo de dígitos que desplegará, además del punto decimal y el signo +, si están presentes. En el siguiente programa se demuestra el proceso al crear una tabla de potencias de 2 y 3. Observe que las columnas se alinean.

```
// Alinea columnas de datos.

#include <iostream>

using namespace std;

int main()
{

  cout << "Ra\u00a1z | Cuadrado |    Cubo\n";
  for(int i = 1; i < 11; ++i) {
    cout.width(4);
    cout << i << " |";
    cout.width(9);
    cout << i * i << " |";
    cout.width(8);
    cout << i * i * i;
    cout << endl;
  }

  return 0;
}
```

Aquí se muestra la salida:

```
Raíz | Cuadrado |    Cubo
   1 |        1 |       1
   2 |        4 |       8
   3 |        9 |      27
   4 |       16 |      64
   5 |       25 |     125
```

```
 6 |      36 |     216
 7 |      49 |     343
 8 |      64 |     512
 9 |      81 |     729
10 |     100 |    1000
```

Opciones

Hay formas sobrecargadas de **width()** y **fill()**, que se muestran aquí:

char fill() const

streamsize width() const

Estas formas obtienen, pero no cambian, la configuración actual.

Otra manera de establecer un ancho de campo de un flujo y un carácter de relleno consiste en usar los manipuladores de E/S **setw()** y **setfill()**. Se describen en *Use manipuladores de E/S para formar datos*.

Justifique la salida

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>	ios_base	fmtflags setf(fmtflags *marcas*) fmtflags setf(fmtflags *marcas1*,*marcas2*) adjustfield internal left right

Por lo general, la salida se alinea a la derecha como opción predeterminada. Esto significa que cuando un ancho de campo excede el tamaño de los datos, se agrega relleno al principio del campo para lograr el ancho deseado. (Consulte la solución anterior para conocer detalles sobre ancho de campo y carácter de relleno.) Puede cambiar este comportamiento al establecer la marca de formato **left** o **internal**. Puede regresar a la justificación a la derecha al establecer la marca **right**. En esta solución se muestra el proceso.

Paso a paso

Para establecer la justificación se necesitan estos pasos:

1. Para dar salida justificada a la izquierda, limpie las marcas especificadas por **adjustfield** y luego establezca la marca **left**.
2. Para dar salida justificada a la derecha, limpie las marcas especificadas por **adjustfield** y luego establezca la marca **right**.
3. Para usar relleno interno para justificar valores numéricos, limpie las marcas especificadas por **adjustfield** y luego establezca la marca **internal**.

Análisis

Hay tres marcas de formato que afectan a la justificación: **right**, **left** e **internal**. De manera colectiva, puede hacer referencia a estas marcas con el valor **adjustfield**. En general, sólo una de estas marcas debe establecerse a la vez. Por tanto, cuando se cambia el método de justificación, debe habilitar la marca que quiera y asegurarse de que las otras dos marcas están deshabilitadas. Esto se hace de manera fácil al usar la forma de dos argumentos **setf()** y el valor **adjustfield**. Verá un ejemplo de esto en breve (consulte *Acceda a las marcas de formato mediante funciones miembro de flujo* para conocer una descripción del establecimiento de las marcas de formato con **setf()**).

Como regla general, la salida está justificada a la derecha, como opción predeterminada. Esto significa que si el ancho de campo es más largo que los datos, el relleno se presentará a la izquierda de los datos. Por ejemplo, considere esta secuencia:

```
cout << 12345678 << endl;
cout.width(8);
cout << "prueba" << endl;
```

Producirá la siguiente salida:

```
12345678
  prueba
```

Cuando se da salida a la cadena "prueba" en un campo que tiene ocho caracteres de largo, se rellena con cuatro caracteres a la izquierda, como se muestra en la salida.

Para especificar justificación a la izquierda, establecemos la marca de formato **left**, como se muestra en esta secuencia:

```
cout.setf(ios::left, ios::adjustfield);
cout << 12345678 << endl;
cout.width(8);
cout << "prueba" << "|" << endl;
```

Produce esta salida:

```
12345678
prueba  |
```

Como puede ver, el relleno se agrega a la derecha de los datos, en lugar de hacerlo a la izquierda. Esto hace que los datos se alineen a la izquierda. Observe cómo la marca **left** está establecida para usar la forma de dos argumentos de **setf()**. Primero limpia todas las marcas a las que hace referencia **adjustfield** y luego establece la marca **left**. Esto asegura que sólo ésta quede establecida.

Cuando se da salida a datos numéricos, puede hacer que se añadan caracteres de relleno dentro de partes del formato al habilitar la marca **internal**. Por ejemplo, si habilita la marca **showpos** (que causa que se muestre un signo **+** en valores positivos), entonces cualquier carácter de relleno se presentará entre el signo **+** y los dígitos.

Ejemplo

En el siguiente programa se muestran las marcas de formato de justificación.

```
// Demuestra las marcas de formato left, right e internal.

#include <iostream>

using namespace std;
```

```
 int main()
{

  // Usa el ancho predeterminado.
  cout << "Formato predeterminado.\n";
  cout << "|";
  cout << 123.45 << "|" << "\n\n";

  // Usa la justificación a la derecha predeterminada
  cout << "Justifica a la derecha en un campo con ancho de 12.\n";
  cout << "|";
  cout.width(12);
  cout << 123.45 << "|" << "\n\n";

  // Cambia a justificación a la izquierda.
  cout << "Justifica a la izquierda en un campo con ancho de 12.\n";
  cout.setf(ios::left, ios::adjustfield);
  cout << "|";
  cout.width(12);
  cout << 123.45 << "|" << "\n\n";

  // Habilita showpos, usa justificación a la izquierda.
  cout << "Habilitando la marca showpos.\n";
  cout.setf(ios::showpos);
  cout << "Justifica a la izquierda en un campo con ancho de 12, otra vez.\n";
  cout << "|";
  cout.width(12);
  cout << 123.45 << "|" << "\n\n";

  // Ahora, usa internal.
  cout << "Habilita la justificaci\u00a2n interna.\n";
  cout.setf(ios::internal, ios::adjustfield);
  cout << "Justificaci\u00a2n interna, en un campo con ancho de 12.\n";
  cout << "|";
  cout.width(12);
  cout << 123.45 << "|" << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Formato predeterminado.
|123.45|

Justifica a la derecha en un campo con ancho de 12.
|      123.45|

Justifica a la izquierda en un campo con ancho de 12.
|123.45      |

Habilitando la marca showpos.
Justifica a la izquierda en un campo con ancho de 12, otra vez.
|+123.45     |
```

```
Habilita la justificación interna.
Justificación interna, en un campo con ancho de 12.
|+     123.45|
```

Opciones

Puede establecer el modo de justificación mediante el uso de los manipuladores de E/S **left**, **right** e **internal**. Se describen en *Use los manipuladores de E/S para formar datos*.

Use los manipuladores de E/S para formar datos

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>		endl fixed left right scientific showpoint showpos
<iomanip>		resetiosflags(ios_base::fmtflags *marcas*) setprecision(int *prec*) setw(int *a*)

C++ combina un conjunto extenso de manipuladores de E/S que le permiten incrustar directivas de formato en una expresión de E/S. Los manipuladores se usan para establecer o limpiar las marcas de formato relacionadas con un flujo. También le permiten especificar el ancho de campo, la precisión y el carácter de relleno. Por tanto, duplican la funcionalidad proporcionada por las funciones miembro del flujo, proporcionando una opción conveniente que le permite escribir código más compacto.

Hay varios manipuladores diferentes definidos por C++. En esta solución se presenta cómo usar una muestra representativa. Debido a que todos los manipuladores trabajan del mismo modo básico, las técnicas presentadas aquí se aplican a todos los manipuladores.

Paso a paso

Para usar un manipulador de E/S se requieren estos pasos:

1. Para usar un manipulador con parámetros, incluya el encabezado **<iomanip>**. Casi todos los manipuladores con parámetros están definidos por **<ios>**, que suele incluirse con otro encabezado de E/S, como **<iostream>**.
2. Para invocar a un manipulador, incruste su nombre dentro de la expresión de salida. Si el manipulador toma un argumento, entonces especifique ese argumento entre paréntesis. De otra manera, simplemente use el nombre del manipulador sin paréntesis.

Análisis

Hay dos tipos básicos de manipuladores de E/S: con parámetros y sin parámetros. Empezaremos con los segundos. Aquí se muestran los manipuladores sin parámetros que operan en flujos de salida:

Manipulador	Propósito
boolalpha	Habilita la marca **boolalpha**.
endl	Da salida a una nueva línea.
ends	Da salida a null.
dec	Habilita la marca **dec**. Deshabilita las marcas **hex** y **oct**.
fixed	Habilita la marca **fixed**. Deshabilita la marca **scientific**.
flush	Limpia el flujo.
hex	Habilita la marca **hex**. Deshabilita las marcas **dec** y **oct**.
internal	Habilita la marca **internal**. Deshabilita las marcas **left** y **right**.
left	Habilita la marca **left**. Deshabilita las marcas **right** e **internal**.
nobootalpha	Deshabilita la marca **noboolalpha**.
noshowbase	Deshabilita la marca **noshowbase**.
noshowpoint	Deshabilita la marca **noshowpoint**.
noshowpos	Deshabilita la marca **noshowpos**.
nounitbuf	Deshabilita la marca **nounitbuf**.
nouppercase	Deshabilita la marca **nouppercase**.
oct	Habilita la marca **oct**. Deshabilita las marcas **dec** y **hex**.
right	Habilita la marca **right**. Deshabilita las marcas **left** e **internal**.
scientific	Habilita la marca **scientific**. Deshabilita la marca **fixed**.
showbase	Habilita la marca **showbase**.
showpoint	Habilita la marca **showpoint**.
showpos	Habilita la marca **showpos**.
unitbuf	Habilita la marca **unitbuf**.
uppercase	Habilita la marca **uppercase**.

Casi todos los manipuladores están declarados en el encabezado **<ios>** (que se incluye automáticamente en otros encabezados, como **<iostream>**). Sin embargo, **endl**, **ends** y **flush** se declaran en **<iostream>**.

Los manipuladores de salida sin parámetros controlan el establecimiento de las diversas marcas de formato. Por ejemplo, para habilitar la marca **showpoint**, se usa el manipulador **showpoint**. Para deshabilitar esta marca, se usa el **noshowpoint**. Observe que los manipuladores que controlan la base del número, la justificación y el formato de punto flotante seleccionan automáticamente el formato especificado, deshabilitando las otras marcas del grupo. Por ejemplo, el manipulador **hex** habilita automáticamente la marca **hex** y deshabilita las marcas **dec** y **oct**. Por tanto, para seleccionar salida hexadecimal, simplemente debe incluir el manipulador **hex**. Las marcas **dec** y **oct** se limpian automáticamente.

Para usar un manipulador con parámetros, debe incluir **<iomanip>**. Define los siguientes manipuladores:

resetiosflags(ios_base::fmtflags *m*)	Deshabilita las marcas especificadas en *m*.
setbase (int *base*)	Establece la base del número en *base*.
setfill(int *car*)	Establece el carácter de relleno en *car*.
setiosflags(ios_base::fmtflags *m*)	Habilita las marcas especificadas en *m*.
setprecision(int *p*)	Establece el número de dígitos de precisión.
setw(int *a*)	Establece el ancho de campo en *a*.

Por ejemplo, para establecer el ancho de campo en 20, incruste **setw(20)** en la expresión de salida. Como en el caso con la función **width()**, **setw** afecta sólo al ancho del siguiente elemento al que habrá de darse salida. Puede usar **setiosflags()** y **resetiosflags()** para establecer o limpiar cualquier combinación arbitraria de marcas.

Los manipuladores de E/S están incrustados en una expresión de E/S. Por ejemplo:

```
cout << setprecision(8) << left << 123.23;
```

Esto establece la precisión en 8, habilita la marca de justificación a la izquierda y luego da salida al número 123.23.

Aunque los manipuladores proporcionan la misma funcionalidad que las funciones miembro **setf()**, **unsetf()**, **width()**, **precision()** y **fill()** descritas en las soluciones anteriores, lo hacen de manera más definida. Por ejemplo, considere esta expresión:

```
cout << setw(12) << fixed << showpos << 98.6 << setw(10) << avg;
```

En una sola línea, establece el ancho de campo en 12, habilita las marcas **fixed** y **showpos** y luego da salida al número 98.6. En seguida, establece el ancho de campo en 10 y da salida al valor de **avg**. El mismo resultado puede obtenerse al usar las funciones miembro de flujo, pero de forma menos compacta:

```
cout.width(12);
cout.setf(ios::fixed, ios::floatfield);
cout.setf(showpos);
cout << 98.6;
cout.width(10);
cout << avg;
```

Ejemplo

En el siguiente ejemplo se muestran varios de los manipuladores de E/S en acción:

```
// Demuestra varios manipuladores de E/S.

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
  cout << "Formato predeterminado: " << 123.123456789 << endl;

  cout << "Formato fijo con precisi\u00a2n de 7: ";
  cout << setprecision(7) << fixed << 123.123456789 << endl;

  cout << "Formato cient\u00a1fico con precisi\u00a2n de 7: ";
  cout << scientific << 123.123456789 << endl;

  cout << "Regresa al formato predeterminado: ";
  cout << resetiosflags(ios::floatfield) << setprecision(6)
       << 123.123456789 << "\n\n";

  cout << "Usa un ancho de campo de 20:\n";
  cout << "|" << setw(20) << "Probando" << "|\n\n";
  cout << "Usa un ancho de campo de 20 con justificaci\u00a2n a la izquierda:\n";
  cout << "|" << setw(20) << left << "Probando" << "|\n\n";

  cout << "Regresando a la justificaci\u00a2n a la derecha.\n\n" << right;

  cout << "Booleanos en ambos formatos: ";
  cout << true << " " << false << " " << boolalpha
       << true << " " << false << "\n\n";

  cout << "Predeterminado: " << 10.0 << endl;
  cout << "Tras establecer las marcas showpos y showpoint: ";
  cout << showpos << showpoint  <<  10.0 << "\n\n";

  cout << "El manipulador setw es muy \u00a3til cuando deben especificarse\n"
       << "anchos de campo repetidos. Por ejemplo:\n";
  cout << setw(8) << "He" << endl <<  setw(8) << "aqu\u00a1" << endl
       << setw(8) << "una" << endl << setw(8) << "columna" << endl
       << setw(8) << "de" << endl << setw(8) << "palabras";

  return 0;
}
```

Aquí se muestra la salida:

```
Formato predeterminado: 123.123
Formato fijo con precisión de 7: 123.1234568
Formato científico con precisión de 7: 1.2312346e+002
Regresa al formato predeterminado: 123.123
```

```
Usa un ancho de campo de 20:
|            Probando|

Usa un ancho de campo de 20 con justificación a la izquierda:
|Probando            |

Regresando a la justificación a la derecha.

Booleanos en ambos formatos: 1 0 true false

Predeterminado: 10
Tras establecer las marcas showpos y showpoint: +10.0000

El manipulador setw es muy útil cuando deben especificarse
anchos de campo repetidos. Por ejemplo:
      He
    aquí
     una
 columna
      de
palabras
```

Opciones

Puede establecer las marcas de formato al hacer llamadas específicas a **setf()** en el flujo. Puede establecer el ancho, la precisión y el carácter de relleno al llamar a **width()**, **precision()** y **fill()** en el flujo. Este método se describió en las soluciones anteriores.

Tiene la opción de crear sus propios manipuladores. Las técnicas necesarias para hacerlo se describen en el capítulo 5.

Forme valores numéricos para una configuración regional y de idioma

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>	ios_base	locale imbue(const &locale *nuevoloc*)
<locale>	locale	

Cuando se da salida a un flujo con valores numéricos, están formados automáticamente con la faceta **num_put** definida por la configuración regional y de idioma actual del flujo. Por tanto, es fácil formar un valor numérico para una configuración específica: simplemente cambie la configuración del flujo por el deseado. La faceta **num_put** para la nueva configuración se usará automáticamente. En esta solución se muestra el proceso.

Paso a paso

Para formar números en relación con una configuración regional y de idioma específica se necesitan estos pasos:

1. Cree un objeto de **locale** que representa la configuración regional y de idioma deseada.
2. Asigne la configuración del flujo creada en el paso 1 al llamar a **imbue()**.

Análisis

Las instrucciones para establecer una configuración regional y de idioma del flujo se presentan en *Obtenga o establezca una configuración regional y de idioma de flujo*, en el capítulo 5. Aquí se presenta un resumen.

La configuración actual define varios aspectos de un formato numérico, incluidos los caracteres usados para el punto decimal y el separador de miles. Como regla general, la configuración predeterminada es "C". Esta configuración regional define un entorno estándar C/C++, que usa el punto como punto decimal y proporciona escasas opciones adicionales de formato. En el caso de muchas aplicaciones, la configuración predeterminada es adecuada. Sin embargo, en casos en que quiera que se desplieguen valores numéricos en un formato compatible con la configuración regional y de idioma del usuario, necesitará especificarla de manera explícita.

Una manera de construir una instancia de **locale** consiste en usar este constructor:

explicit locale(const char **nombre*)

Aquí, *nombre* especifica el nombre de la configuración regional y de idioma, como German, Spanish_Spain o US. Si *nombre* no representa una configuración válida, entonces se lanza la excepción **runtime_error**. Lo que constituye un nombre válido de configuración puede variar (y seguramente lo hará) entre compiladores. Los ejemplos mostrados en este libro funcionan con Microsoft Visual C++ y tal vez funcionarán con otros compiladores, pero debe consultar la documentación de su compilador para conocer más detalles.

Para establecer la configuración regional y de idioma del flujo, llame a **imbue()** en el flujo. Aquí se muestra:

locale imbue(const locale &*nuevoloc*)

La configuración del flujo que invoca se establece en *nuevoloc*, y se devuelve el anterior.

Ejemplo

En el siguiente ejemplo se muestra la manera en que diferentes configuraciones regionales y de idioma afectan al formato de los números. El programa empieza por desplegar un valor en el formato predeterminado (que suele determinarse mediante la configuración regional y de idioma de C). Luego especifica la configuración English, y despliega el mismo valor. Por último, usa la configuración Spanish_Spain. Observe que en English, el separador de miles es la coma y el punto decimal es un punto. En Spanish_Sapin, esto es al revés: el separador de miles es el punto y el punto decimal es la coma. Además, tome nota de que están establecidas la precisión y la marca **fixed**, pero no se ven afectadas por la configuración regional y de idioma.

```
// Formato de valores numéricos con una configuración regional
// y de idioma específica.
```

```
#include <iostream>
#include <locale>
#include <iomanip>

using namespace std;

int main()
{
  // Usa un formato fijo con 2 lugares decimales.
  cout << fixed << setprecision(2);

  cout << "Formato predeterminado: " << 12345678.12 << "\n\n";

  // Establece la configuración regional y de idioma en English.
  locale eloc("English");
  cout.imbue(eloc);

  cout << "Formato English: " << 12345678.12 << "\n\n";

  locale sloc("Spanish_Spain");
  cout.imbue(sloc);

  cout << "Formato Spanish: " << 12345678.12 << "\n\n";
  return 0;
}
```

Aquí se muestra la salida:

```
Formato predeterminado: 12345678.12

Formato English: 12,345,678.12

Formato Spanish: 12.345.678,12
```

Opciones

Tiene la opción de formar valores numéricos en un formato monetario al usar la faceta **money_put**. Usa automáticamente la configuración regional y de idioma actual. Consulte *Forme valores monetarios usando la faceta **money_put*** para conocer más detalles.

Aunque en el ejemplo anterior, y en muchos de los ejemplos de este capítulo, se usa **cout** como flujo de destino, el mismo método básico funciona con todos los flujos de salida. Por ejemplo, la siguiente secuencia crea un **ofstream** llamado **archsalida** y lo conecta con un archivo llamado **prueba.dat**. Luego habilita la marca **fixed** y establece la precisión en 2. A continuación, establece la configuración regional y de idioma en Spanish_Spain. Por último, da salida a 12345678.12 a **archsalida**.

```
ofstream archsalida("prueba.dat");
archsalida.imbue(locale("Spanish_Spain"));
archsalida << fixed << setprecision(2);
archsalida << 12345678.12;
```

Después de que se ejecute esta secuencia, **prueba.dat** contendrá lo siguiente:

```
12.345.678,12
```

Como observará, está formado para español de España.

Aunque el uso del operador de E/S **<<** es la manera más fácil (y, con frecuencia, la mejor) para formar salida numérica, puede usar directamente la faceta **money_put**. Esto se hace al obtener

primero una referencia a la faceta **num_put** para la configuración regional y de idioma actual al llamar a **use_facet()**, que se describe en *Revisión general de las facetas*, casi al principio de este capítulo. Luego, usando esta referencia, llame a **put()** para formar un valor y darle salida a un flujo.

La faceta **num_put** se declara así:

```
template <class CharT, class OutItr = ostreambuf_iterator<CharT> >
  class num_put : public locale::facet { // ...
```

CharT especifica el tipo de caracteres sobre el que se opera. **OutItr** especifica el tipo de iterador que se utiliza para escribir datos formados. Observe que la opción predeterminada es **ostreambuf_iterator**.

La función **put()** definida por **num_put** tiene varias versiones. He aquí una. Forma un valor **double**:

```
iter_type put(iter_type itr_flujo, ios_base &flujo,
              char_type carrelleno, double val) const
```

Un iterador al flujo de salida se pasa en *itr_flujo*. El tipo **iter_type** es un **typedef** para el tipo de iterador. Como opción predeterminada, tipo es **ostreambuf_iterator**. Hay una conversión automática a este tipo desde cualquier objeto de **basic_ostream**, de modo que, por lo general, simplemente pasará el flujo sobre el que se está actuando. Se pasa una referencia al flujo de salida en *flujo*. Sus configuraciones de marca, precisión y ancho se usan para determinar el formato. El carácter de relleno se pasa en *carrelleno*. El valor que habrá de formarse se pasa en *val*.

Al unir todas las piezas, la siguiente secuencia utiliza **num_put** para desplegar el número 1024.256 en formato fijo, con una precisión de 2 y un ancho de 20, en la configuración regional y de idioma actual.

```
cout << fixed << setprecision(2) << setw(20);
const num_put<char> &np = use_facet<num_put<char> >(cout.getloc());
np.put(cout, cout,1 1, 1024.256);
```

Como observará, esto requiere mucho más esfuerzo que cuando se usa el operador **<<** y no se gana nada con eso.

Tiene la opción de leer un número de manera sensible a la configuración regional y de idioma utilizando **num_get**. Define la función **get()** que lee un número en su forma de flujo.

Forme valores monetarios empleando la faceta money_put

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ios>	ios_base	locale getloc() const
<ios>	ios	locale imbue(const &locale *nuevoloc*)
<locale>	locale	template <class facet> const Facet &use_facet(const locale &*loc*)
<locale>	money_put	iter_type put(iter_type *itr_flujo*, bool *sim_mon_int*, ios_base &*flujo*, char_type *carrelleno*, long double *val*) const

En cuanto a la formación, tal vez la pregunta de tipo "¿Cómo hacer?" más frecuente sea "¿Cómo despliego valores monetarios?". Debido a que el formato numérico predeterminado no está diseñado para este fin, el método apropiado es fuente de mucha confusión. Por fortuna, la solución es muy simple: use la faceta **money_put** definida por la biblioteca de localización de C++. Al hacerlo así, se produce automáticamente el formato correcto para la configuración regional y de idioma actual. En esta solución se muestra el proceso.

Paso a paso

Para desplegar un valor monetario mediante la faceta **money_put** se necesitan estos pasos:

1. Construya un objeto de **locale** que represente la configuración regional y de idioma con que se formará el valor monetario.
2. Establezca la configuración al llamar a **imbue()** en el flujo que estará recibiendo la salida formada. Pase **imbue()** al objeto de **locale** del paso 1.
3. Obtenga la faceta **money_put** al llamar a **use_facet()**, especificando la configuración regional y de idioma de la que obtendrá la faceta. En general, será la configuración actual empleada por el flujo de salida. Puede obtenerla al llamar a **getloc()** en el flujo.
4. Forme los datos al llamar a **put()** en el objeto devuelto por **use_facet()**, especificando el flujo en que se escribirá la salida.

Análisis

Una revisión general del subsistema de localización de C++ se presentó cerca del principio de este capítulo. Las funciones **imbue()** y **getloc()** se describieron en *Obtenga o establezca la configuración regional y de idioma de un flujo*, en el capítulo 5. También se presentó un resumen del método **imbue()** en la solución anterior. Recuerde que **imbue()** establece la configuración regional y de idioma de un flujo.

La faceta **money_put** se declara como se muestra a continuación:

```
template <class CharT, class OutItr = ostreambuf_iterator<CharT> >
  class money_put : public locale::facet { // ...
```

CharT especifica el tipo de caracteres sobre los que se opera. **OutItr** especifica el tipo de iterador que se utiliza para escribir datos formados. Observe que la opción predeterminada es **ostreambuf_iterator**.

Para obtener la faceta **money_put**, debe llamar a **use_facet()**. Esta función se describió en *Revisión general de las facetas*, casi al principio de este capítulo. Recuerde que es una función genérica global definida por **<locale>**, con el siguiente prototipo:

```
template <class Facet> const Facet &use_facet(const locale &loc)
```

El parámetro de la plantilla **Facet** especifica la faceta, que será **money_put** en este caso. La configuración regional y de idioma se pasa mediante *loc*. Se devuelve una referencia a la faceta. Por tanto, **use_facet()** obtiene una versión específica de la faceta adecuada para la configuración. Se lanza una excepción **bad_cast** si la faceta deseada no está disponible. En general, las facetas predefinidas, incluida **money_put**, estarán disponibles.

Por lo general, la instancia de **locale** pasada a **use_facet()** será la usada por el flujo de salida al que se aplicará la faceta. Puede obtener la configuración regional y de idioma actual de un flujo al llamar a **getloc()** en el flujo. Aquí se muestra cómo:

```
locale getloc() const
```

Devuelve el objeto de **locale** asociado con el flujo.

Con el uso de la faceta devuelta por **use_facet()**, puede formar un valor monetario al llamar a **put()**. Tiene dos formas. Aquí se muestra la usada en esta solución:

iter_type put(iter_type *itr_flujo*, bool *sim_mon_int*, ios_base &*flujo*,
char_type *carrelleno*, long double *val*) const

En *itr_flujo* se pasa un iterador al flujo de salida. El tipo **itr_type** es un **typedef** para el tipo de iterador. Como opción predeterminada, este tipo es **ostreambuf_iterator**. Se hace una conversión automática a este tipo desde cualquier objeto de **basic_ostream**, de modo que por lo general pasará el flujo sobre el que se actuará. Si el símbolo monetario habrá de mostrarse en su forma internacional, pase true a *sim_mon_int*. Pase false para usar el símbolo local. Pase una referencia al flujo de salida en *flujo*. Si está establecida una marca **showbase**, entonces se mostrará el símbolo monetario. El carácter de relleno se pasa en *charrelleno*. El valor que habrá de formarse se pasa en *val*. La función **put()** devuelve un iterador que señala una posición después del último carácter al que se da salida.

La única peculiaridad asociada con **money_put** es que opera sobre datos que no contienen un punto decimal. Por ejemplo, el valor 1724.89 se pasa a **put()** como 172489. El formador monetario agrega la coma y el punto decimal. Para el caso de dólares estadounidenses, se transforma en 1,724.89. Si ha habilitado el símbolo monetario doméstico, entonces el resultado es $1,724.89.

Ejemplo

En el siguiente ejemplo se muestra cómo usar **money_put**.

```
// Usa money_put para dar salida a valores monetarios.

#include <iostream>
#include <locale>

using namespace std;

int main()
{
  double saldo = 5467.87;

  locale euloc("English_US");
  locale sloc("Spanish_Spain");

  // Establece la marca showbase para desplegar el símbolo monetario.
  cout << showbase;

  cout << "Formato monetario para d\u00a2lares de Estados Unidos:\n";
  cout.imbue(euloc);
  const money_put<char> &mon_eu =
          use_facet<money_put<char> >(cout.getloc());

  mon_eu.put(cout, false, cout, ' ', "123456");
  cout << endl;
  mon_eu.put(cout, true, cout, ' ', -299);
  cout << endl;
  mon_eu.put(cout, false, cout, ' ', saldo * 100);
  cout << "\n\n";
```

```
  cout << "Ahora muestra el monto en el formato internacional Spanish_Spain:\n";
  cout.imbue(sloc);
  const money_put<char> &mon_s =
          use_facet<money_put<char> >(cout.getloc());

  mon_s.put(cout, true, cout, ' ', 123456);
  cout << endl;
  mon_s.put(cout, true, cout, ' ', -299);
  cout << endl;
  mon_s.put(cout, true, cout, ' ', saldo * 100);

  return 0;
}
```

Aquí se muestra la salida:

```
Formato monetario para dólares de Estados Unidos:
$1,234.56
USD-2.99
$5,467.87

Ahora muestra el monto en el formato internacional Spanish_Spain:
EUR1.234,56
EUR-2,99
EUR5.467,87
```

Opciones

Hay una segunda forma de **put()** que da formato a una versión de cadena del valor. Se muestra a continuación:

iter_type put(iter_type *itr_flujo*, bool *sim_mon_int*, ios_base &*marcasflujo*,
char_type *carrelleno*, long double *valcad*) const

Funciona igual que la primera versión, excepto que el valor que habrá de formarse se pasa como una cadena en *valcad*.

Como se explicó, si solicita una faceta que no está disponible, entonces se lanza una excepción **bad_cast**. Para evitar esta posibilidad, puede determinar si una faceta está disponible para una configuración regional y de idioma dada al llamar a **has_facet()**. Se trata de una función de plantilla global definida por **<locale>**. Aquí se muestra:

template <class facet> bool has_facet(const locale &*loc*) throw()

Devuelve true si la faceta especificada está disponible y false, de lo contrario. En general, las facetas estándar siempre estarán disponibles, pero es probable que las personalizadas no lo estén. En cualquier caso, tal vez quiera usar **has_facet()** para confirmar que puede usarse una faceta. Al hacerlo así puede evitar una excepción.

Tiene la opción de leer valores monetarios formados al usar la faceta **money_get**. Define la función **get()**, que lee un valor monetario en su forma de cadena.

Use las facetas moneypunct y numpunct

Componentes clave		
Encabezados	**Clases**	**Funciones**
<locale>	moneypunct	string_type cur_symbol() const char_type decimal_point() const int frac_digits() const char_type thousands_sep() const string grouping() const
<locale>	numpunct	char_type decimal_point() const char_type thousands_sep() const string grouping() const

Aunque la formación de valores numéricos mediante **num_put** y de valores monetarios mediante **money_put** suele ser la mejor opción, tiene la opción de tomar el control del proceso, si lo desea. La clave está en obtener los signos de puntuación y las reglas usadas para formar valores monetarios y numéricos relacionados con una configuración regional y de idioma. Estos signos de puntuación son el símbolo monetario, el separador de miles y el punto decimal. Las reglas son el número de dígitos fraccionales desplegados y el número de dígitos en un grupo. Ambos están disponibles mediante las facetas **moneypunct** y **numpunct**. En esta solución se muestra cómo obtenerlas.

Paso a paso

Para usar la faceta **numpunct** se necesitan estos pasos:

1. Obtenga la faceta **numpunct** para una configuración regional y de idioma específica al llamar a **use_facet()**. Utilice esta faceta para obtener la puntuación numérica y las reglas de la configuración, como se describe en los pasos siguientes.
2. Obtenga el carácter de punto decimal al llamar a **decimal_point()**.
3. Obtenga el separador de miles al llamar a **thousands_sep()**.
4. Obtenga la regla que determina la agrupación de dígitos al llamar a **grouping()**.

Para usar la faceta **moneypunct** se necesitan estos pasos:

1. Obtenga la faceta **moneypunct** para una configuración regional y de idioma específica al llamar a **use_facet()**. Utilice esta faceta para obtener la puntuación numérica y las reglas de la configuración, como se describe en los pasos siguientes.
2. Obtenga el símbolo de moneda al llamar a **cur_symbol()**.
3. Obtenga el carácter de punto decimal al llamar a **decimal_point()**.
4. Obtenga el separador de miles al llamar a **thousands_sep()**.

5. Obtenga el número de dígitos fraccionales usados para representar valores monetarios al llamar a **frac_digits()**.
6. Obtenga la regla que determina la agrupación de dígitos al llamar a **grouping()**.

Análisis

Los signos de puntuación y las reglas para valores numéricos están encapsulados dentro de la faceta **numpunct**. Se declara como se muestra a continuación:

```
template <class CharT> class numpunct : public locale::facet { // ...
```

CharT especifica el tipo de caracteres sobre el que se operará. Como todas las facetas, hereda **locale::facet**.

Puede obtener una referencia a una faceta **numpunct** al llamar a **use_facet()**, especificando **numpunct** como la faceta que habrá de obtenerse. La función **use_facet()** está definida globalmente por **<locale>**, como se describió en *Revisión general de las facetas*. En la siguiente secuencia se muestra cómo usarla para obtener una faceta **numpunct** para la configuración regional y de idioma usada por **cout**:

```
const numpunct<char> &numpunct = use_facet<numpunct<char> >(cout.getloc());
```

Dada una referencia a la faceta **numpunct**, puede obtener los diversos signos de puntuación y las reglas que se relacionan con valores numéricos. Cada valor está modificado de acuerdo con la configuración regional y de idioma de la faceta. Estos elementos están disponibles mediante funciones. A continuación se muestran las usadas en esta solución:

Función	Descripción
char_type decimal_point() const	Devuelve el carácter usado como punto decimal.
char_type thousands_sep() const	Devuelve el carácter usado para separar (es decir, agrupar) miles.
string grouping() const	Devuelve las reglas que definen las agrupaciones de dígitos.

Aquí, **char_type** es una **typedef** para el tipo de carácter, que será **char** para flujos de **char**.

Los signos de puntuación y las reglas para valores monetarios están encapsulados dentro de la faceta **moneypunct**. Se declara como se muestra a continuación:

```
template <class CharT, bool Intl = false>
  class moneypunct : public locale::facet, public money_base { // ...
```

CharT especifica el tipo de caracteres sobre el que se operará. El tipo **Intl** indica si se usarán formatos internacionales o locales. La opción predeterminada es local. Como todas las facetas, hereda **locale::facet**. La clase **money_base** define aspectos de los formatos monetarios que son dependientes de los parámetros de tipo. Se describen más a fondo en la secuencia *Opciones* de esta solución.

Como en el caso de **numpunct**, se obtiene una referencia a **moneypunct** al llamar a **use_facet()**. He aquí un ejemplo:

```
const moneypunct<char> &us_moneypunct = use_facet<moneypunct<char>
                                         >(cout.getloc());
```

Esta instrucción obtiene la faceta **moneypunct** para la configuración regional y de idioma de **cout**.

Dada una referencia a la faceta **moneypunct**, puede obtener los diversos signos de puntuación y las reglas que se relacionan con valores numéricos al llamar a funciones mediante la referencia. Cada valor está modificado de acuerdo con la configuración regional y de idioma de la faceta. Aquí se muestran las usadas en esta solución:

Función	Descripción
string_type cur_symbol() const	Devuelve el carácter o los caracteres usados como símbolo monetario.
char_type decimal_point() const	Devuelve el carácter usado como punto decimal.
int frac_digits() const	Devuelve el número de dígitos fraccionales que suelen desplegarse para valores monetarios.
char_type thousands_sep() const	Devuelve el carácter usado para separar (es decir, agrupar) miles.
string grouping() const	Devuelve las reglas que definen las agrupaciones de dígitos.

Aquí, **char_type** es una **typedef** para el tipo de carácter, que será **char** para flujos de **char** y **string_type** es un **typedef** para el tipo de **string**, que será **string** para los flujos de **char**.

El valor devuelto por **grouping()** es el mismo para **numpunct** y **moneypunct**. Es un valor de cadena en que el valor de unicode de cada carácter representa el número de dígitos en un grupo, yendo de derecha a izquierda, y empezando con el primer grupo a la izquierda del punto decimal. Si el tamaño del grupo no está especificado, se utiliza el tamaño del grupo anterior. Por tanto, si todos los tamaños son iguales, entonces sólo se especificará un valor. Recuerde que el que se usa es el valor de unicode del carácter, no su dígito legible para los seres humanos. Por tanto, el carácter '\003' (no '3') representa tres dígitos.

Ejemplo

En el siguiente ejemplo se muestra cómo usar **moneypunct** y **numpunct** para obtener los signos de puntuación y las reglas de agrupamiento para Estados Unidos:

```
// Demuestra signos de puntuación y agrupaciones monetarias y numéricas.

#include <iostream>
#include <locale>

using namespace std;

int main()
{
  // Crea una configuración regional y de idioma para US English.
  locale usloc("English_US");

  // Establece la configuración regional y de idioma para US English.
  cout.imbue(usloc);

  // Obtiene una faceta moneypunct para cout.
  const moneypunct<char> &us_monpunct =
          use_facet<moneypunct<char> >(cout.getloc());
```

```
  cout << "Puntuaci\u00a2n monetaria para EU:\n";
  cout << "  S\u00a1mbolo de moneda: " << us_monpunct.curr_symbol() << endl;
  cout << "  Punto decimal: " << us_monpunct.decimal_point() << endl;
  cout << "  Separador de miles: " << us_monpunct.thousands_sep() << endl;
  cout << "  D\u00a1gitos de fracci\u00a2n: " << us_monpunct.frac_digits() << endl;

  cout << "  N\u00a3mero de reglas de agrupaci\u00a2n: "
       << us_monpunct.grouping().size() << endl;

  for(unsigned i=0; i < us_monpunct.grouping().size(); ++i)
    cout << "  Tama\u00a4o del grupo " << i << ": "
         << (int)us_monpunct.grouping()[0] << endl;

  cout << endl;

  // Obtiene una faceta numpunct para cout.
  const numpunct<char> &us_numpunct =
          use_facet<numpunct<char> >(cout.getloc());

  cout << "Puntuaci\u00a2n de n\u00a3meros para EU:\n";
  cout << "  Punto decimal: " << us_monpunct.decimal_point() << endl;
  cout << "  Separador de miles: " << us_monpunct.thousands_sep() << endl;

  cout << "  N\u00a3mero de reglas de agrupaci\u00a2n: "
       << us_monpunct.grouping().size() << endl;

  for(unsigned i=0; i < us_monpunct.grouping().size(); ++i)
    cout << "  Tama\u00a4o del grupo " << i << ": "
         << (int)us_monpunct.grouping()[0] << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Puntuación monetaria para EU:
  Símbolo de moneda: $
  Punto decimal: .
  Separador de miles: ,
  Dígitos de fracción: 2
  Número de reglas de agrupación: 1
  Tamaño del grupo 0: 3

Puntuación de números para EU:
  Punto decimal: .
  Separador de miles: ,
  Número de reglas de agrupación: 1
  Tamaño del grupo 0: 3
```

Opciones

La faceta **numpunct** define las funciones **truename()** y **falsename()**, que se muestran a continuación:

```
string_type truename() const
```

```
string_type falsename() const
```

Devuelven los nombres para **true** y **false** en relación con la configuración regional y de idioma especificada.

La faceta **moneypunct** le permite obtener los signos usados para indicar valores monetarios positivos y negativos al llamar a las funciones **positive_sign()** y **negative_sign()**, que se muestran aquí:

string_type positive_sign() const

string_type negative_sign() const

Observe que se devuelve una cadena, en lugar de un solo carácter. Esto permite el uso de varios signos.

Con **moneypunct**, también puede obtener patrones que representan los formatos positivos y negativos al llamar a **pos_format()** y **neg_format()**, respectivamente. Aquí se muestran:

pattern pos_format() const

pattern neg_format() const

Cada uno devuelve un objeto de **pattern** que describe el formato indicado.

El tipo **pattern** es una **struct** definida dentro de la clase **money_base**. Ésta es una clase de base para **moneypunct**. Aquí se muestra:

```
class money_base {
public:
  enum part { none, space, symbol, sign, value };
  struct pattern {
    char field[4];
  };
};
```

Cada elemento de **field** contiene un valor **part**. (El C++ estándar establece que se usa una matriz de **char**, en lugar de una de **part**, con **field** "simplemente para obtener mayor eficiencia".) Cada elemento de **pattern** indica cuál parte del formato monetario debe aparecer en ese punto, donde la primera parte es **field[0]**, la segunda **field[1]**, etc. He aquí lo que significa la enumeración de constantes:

none	No hay una salida correspondiente.
space	Un espacio.
symbol	El símbolo de moneda.
sign	El signo positivo o negativo.
value	El valor.

Por ejemplo, suponiendo el programa anterior, la siguiente secuencia despliega el patrón negativo:

```
// Muestra el patrón numérico negativo.
for(int i=0; i < 4; ++i)
  switch(us_monpunct.neg_format().field[i]) {
    case money_base::none: cout << "ninguno ";
      break;
    case money_base::value: cout << "valor ";
      break;
    case money_base::space: cout << "espacio ";
      break;
```

```
    case money_base::symbol: cout << "s\u00a1mbolo ";
      break;
    case money_base::sign: cout << "signo ";
      break;
  }
```

Produce la siguiente salida:

```
signo símbolo valor ninguno
```

Esto indica que un valor monetario negativo empieza con un signo, seguido por el símbolo de moneda y por último el valor.

Forme la fecha y hora con la faceta time_put

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ctime>		struct tm &localtime(const time_t **hora*) time_t time(time_t **apt_h*)
<ios>	ios_base	locale getloc() const
<ios>	ios	locale imbue(const &locale *nuevoloc*)
<locale>	locale	template <class Facet> const Facet &use_face(const locale &*loc*)
<locale>	time_put	iter_type put(iter_type *itr_flujo*, ios_base &*no_usado*, char_type *carrelleno*, const tm **h*, const char_type **inicio_patron*, const char_type **final_patrón*) const

Si "¿Cómo despliego valores monetarios?" es la pregunta de formato más frecuente, la que le sigue en frecuencia es "¿Cómo despliego la hora y la fecha?" Aunque el concepto es fácil, la formación de la hora y la fecha requiere más trabajo del que podría pensar al principio. El problema es doble. En primer lugar, los formatos de fecha y hora son sensibles a la configuración regional y de idioma. Por tanto, no hay un formato universal que funcionará en todos los casos. En segundo lugar, la hora y la fecha pueden desplegarse de muchas maneras. Como resultado, hay muchas opciones para elegir.

En general, hay dos maneras de formar la fecha y la hora usando C++. La primera consiste en llamar a la función **strftime()** de C. Forma la fecha y la hora con base en la configuración regional y de idioma global. (Consulte *Forme la fecha y la hora usando **strftime()*** para conocer más detalles.) El segundo método está definido por C++ y emplea la faceta **time_put** definida por el subsistema de localización. El uso de **time_put** ofrece una ventaja principal: le permite formar la fecha y la hora de acuerdo con la configuración regional y de idioma de un flujo específico, en lugar de aplicar la configuración global usada por **strftime()**. También está integrada con otras facetas de formación de C++, como **money_put**. Por eso, la formación de la fecha y la hora usando **time_put** es el método recomendado para casi todas las aplicaciones. En esta solución se muestra cómo ponerla en acción.

Paso a paso

Para formar la fecha y la hora usando la faceta **time_put** se necesitan estos pasos:

1. Construya un objeto de **locale** que representa la configuración regional y de idioma para la que se han formado la fecha y la hora.
2. Establezca la configuración al llamar a **imbue()** en el flujo que estará recibiendo la salida formada. Pase **imbue()** al objeto de **locale** del paso 1.
3. Obtenga la faceta **time_put** al llamar a **use_facet()**, especificando la configuración regional y de idioma de la que se obtendrá la faceta. En general, será la configuración actual usada por el flujo de salida. Puede obtenerlo al llamar a **getloc()** en el flujo.
4. Obtenga el apuntador **tm** que señala a la hora que se formará. Una manera de obtenerlo consiste en llamar a **localtime()**. Devuelve la hora local proporcionada por el equipo.
5. Forme la fecha y la hora al llamar a **put()** en el objeto devuelto por **use_facet()**, especificando el flujo al que se escribirá la salida.

Análisis

Una revisión general del subsistema de localización de C++ se presentó casi al principio de este capítulo. Las funciones **imbue()** y **getloc()** se describieron en *Obtenga o establezca la configuración regional y de idioma de un flujo*, en el capítulo 5. También se presentó un resumen de los métodos **imbue()** y **getloc()** en las dos soluciones anteriores.

Para formar la fecha y la hora, por lo general usará la faceta **time_put**. Se declara así:

```
template <class CharT, class OutItr = ostreambuf:iterator<CharT> >
  class time_put : public locale::facet { // ...
```

CharT especifica el tipo de caracteres sobre el que se operará. **OutItr** especifica el tipo de iterador que se usa para escribir los datos formados. Observe que la opción predeterminada es **ostreambuf_iterator**.

Para obtener la faceta **time_put**, debe llamar a **use_facet()**. Esta función se describe en *Revisión general de las facetas*, casi al principio de este capítulo. Recuerde que es una función genérica global definida por **<locale>**, con el siguiente prototipo:

```
template <class Facet> const Facet &use_facet(const locale &loc)
```

El parámetro de plantilla **Facet** especifica la faceta, que será **time_put** en este caso. La configuración regional y de idioma se pasa mediante *loc*. Se devuelve una referencia a la faceta. Se lanza una excepción **bad_cast** si la faceta deseada no está disponible. En general, las facetas predefinidas, incluida **time_put**, estarán disponibles.

Con el uso de la faceta **time_put** obtenida de **use_facet()**, puede formar un valor de hora llamando a **put()**. Tiene dos formas. Aquí se muestra la usada en esta solución:

iter_type put(iter_type *itr_flujo*, ios_base &*no_usado*, char_type *carrelleno*,
const tm **h*, const char_type **inicio_patron*,
const char_type **final_patron*) const

Un iterador al flujo de salida se pasa en *itr_flujo*. El tipo **iter_type** es un **typedef** para el tipo de iterador. Como opción predeterminada, el tipo es **ostreambuf_iterator**. Hay una conversión automática a este tipo desde cualquier objeto de **basic_ostream**, de modo que, por lo general, simplemente pasará el flujo sobre el que se está actuando. El parámetro *no_usado* no se usa. (Puede pasar una referencia al flujo de salida como marcador de posición.) El carácter de relleno se pasa en *carrelleno*. Un apuntador a una estructura **tm** que contiene la fecha y la hora se pasa en *t*. Un apuntador al principio de la cadena que define un patrón que se usará para formar la fecha y la hora se pasa en *inicio_patron*. Uno al final de la cadena se pasa en *final_patron*. El tipo **char_type** es un **typedef** para el tipo de carácter. En el caso de cadenas de **char**, que son las que se usan en este libro, este **char_type** es **char**.

La estructura de **tm** está definida en **<time>** y se hereda de C. Muestra lo que se llama la forma "desglosada" de la fecha y la hora. Se presenta a continuación:

```
struct tm {
  int tm_sec;   // segundos, 0-61
  int tm_min;   // minutos, 0-59
  int tm_hour;  // horas, 0-23
  int tm_mday;  // día del mes, 1-31
  int tm_mon;   // meses desde enero, 0-11
  int tm_year;  // años desde 1900
  int tm_wday;  // días desde el domingo, 0-6
  int tm_yday;  // días desde el 1° de enero, 0-365
  int tm_isdst; // Indicador de hora de ahorro de luz del día
}
```

Puede construir un objeto de **tm** al establecer manualmente sus miembros, pero no lo hará con frecuencia. Más a menudo, simplemente obtendrá un objeto de **tm** que contiene la fecha y la hora actuales al usar una función definida por **<ctime>**. La usada por esta solución es **localtime()** y se muestra a continuación:

struct tm *localtime(continuación time_t **hora*)

Toma la hora codificada como un valor **time_t** y devuelve un apuntador a una estructura **tm** que contiene la hora desglosada en sus componentes individuales. La hora está representada en hora local. La estructura **tm** señalada por el apuntador devuelto por **localtime()** está asignada estáticamente y se sobreescribe cada vez que se llama a la función. Si quiere guardar el contenido de la estructura, debe copiarla en otro lugar.

Puede obtener un valor **time_t** de varias maneras. El método usado en esta solución consiste en llamar a **time()**. Es otra función definida por **<ctime>** y obtiene la hora actual del sistema. Se muestra a continuación:

time_t time(time_t **apt_h*)

Devuelve la hora actual del sistema. Esto suele representarse como el número de segundos a partir del 1 de enero de 1970. Si el sistema no tiene hora, se devuelve –1. La función puede llamarse con un apuntador nulo o con uno a una variable de tipo **time_t**. Si se usa el primero, también se asignará a la hora la variable señalada por *apt_t*.

En la función **put()**, la cadena de patrón señalada por *inicio_patron* contiene dos tipos de elementos. El primero son caracteres normales, que simplemente se despliegan como tales. El segundo son especificadores de formato de fecha y hora, que determinan cuáles componentes de fecha y hora se despliegan. Estos especificadores de formato son los mismos que los usados por la función **strftime()** heredada de C. Se presentan en la tabla 6-1. (Consulte *Forme la fecha y hora con* ***strftime()***.) Los especificadores de formato empiezan con un signo de porcentaje (%) y son seguidos por un comando de formato. Por ejemplo, **%H** causa que la hora se despliegue empleando el reloj de 24 horas. **%Y** hace que se muestre el año. Puede combinar caracteres regulares y especificadores de fecha/hora en el mismo patrón. Por ejemplo,

```
char *custom_pat = "La fecha de hoy es %x";
```

Suponiendo que la fecha es 1 de enero de 2009, entonces esto produce la siguiente salida:

```
La fecha de hoy es 1/1/2009
```

Ejemplo

En el siguiente ejemplo se muestra **time_put** en acción. Despliega la fecha y hora en English y Spanish_Spain.

```
// Da salida a la fecha y hora usando la faceta time_put.

#include <iostream>
#include <locale>
#include <cstring>
#include <ctime>

using namespace std;

int main()
{
  // Obtiene la hora actual del sistema.
  time_t t = time(NULL);
  tm *hora_act = localtime(&t);

  // Crea configuraciones regionales y de idioma para US y Spanish_Spain.
  locale usloc("English_US");
  locale sloc("Spanish_Spain");

  // Establece la configuración regional y de idioma para US
  // y obtiene la faceta time_put para US.
  cout.imbue(usloc);
  const time_put<char> &hora_us =
          use_facet<time_put<char> >(cout.getloc());

  // %c especifica el patrón de fecha y hora estándar.
  char *pat_est = "%c";
  char *pat_est_fin = pat_est + strlen(pat_est);
```

```
  // El siguiente patrón personalizado despliega horas y minutos
  // y después muestra la fecha.
  char *pat_przado = "%A %B %d, %Y %H:%M";
  char *pat_przado_fin = pat_przado + strlen(pat_przado);

  cout << "Formato de fecha y hora US est\u00a0ndar: ";
  hora_us.put(cout, cout, ' ', hora_act, pat_est, pat_est_fin);
  cout << endl;

  cout << "Formato de fecha y hora US personalizado: ";
  hora_us.put(cout, cout, ' ', hora_act, pat_przado, pat_przado_fin);
  cout << "\n\n";

  // Establece la configuración regional y de idioma y obtiene
  // la faceta time_put para España.
  cout.imbue(sloc);
  const time_put<char> &hora_g =
          use_facet<time_put<char> >(cout.getloc());

  cout << "Formato de fecha y hora Spanish_Spain est\u00a0ndar: ";
  hora_g.put(cout, cout, ' ', hora_act, pat_est, pat_est_fin);
  cout << endl;

  cout << "Formato de fecha y hora Spanish_Spain personalizado: ";
  hora_g.put(cout, cout, ' ', hora_act, pat_przado, pat_przado_fin);
  cout << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Formato de fecha y hora US estándar: 11/24/2008 3:54:15 PM
Formato de fecha y hora US personalizado: Monday November 24, 2008 15:54

Formato de fecha y hora Spanish_Spain estándar: 24/11/2008 15:54:15
Formato de fecha y hora Spanish_Spain personalizado: lunes noviembre 24, 2008
15:54
```

Opciones

Otra manera de formar la fecha y hora consiste en usar la función **strftime()** heredada del lenguaje C. Si está usando la configuración regional y de idioma global, entonces **strftime()** es un poco más fácil de usar que la faceta **time_put**. Consulte *Forme la fecha y hora con **strftime()*** para conocer más detalles.

Hay una segunda forma de **put()** que le permite determinar un solo especificador de formato de fecha y hora. Aquí se muestra:

iter_type put(iter_type *flujo*, ios_base &*no_usado*, char_type *carrelleno*,
    const tm **h*, char *ftm*, char *modo* = 0) const

Los primeros cuatro parámetros son los mismos que en la primera versión. El especificador de formato se pasa en *fmt*, y un modificador de formato opcional se pasa en *modo*. No todos los entornos dan soporte a modificadores. Si lo tienen, están definidos por la implementación. La función devuelve un iterador a uno después del último carácter escrito.

Forme datos en una cadena

Componentes clave		
Encabezados	**Clases**	**Funciones**
<sstream>	ostringstream	string str() const

En ocasiones, es útil construir de antemano una cadena que contenga salida formada. Así puede darse salida a la cadena cuando sea necesario. Esta técnica resulta especialmente útil cuando se trabaja en un entorno de ventanas, como Windows, en que los datos se despliegan mediante un control. En este caso, a menudo necesitará formar los datos antes de desplegarlos. Esto suele realizarse de manera más fácil en C++ mediante el uso de un flujo de cadena, como **ostringstream**. Debido a que todos los flujos trabajan de la misma manera, las técnicas descritas en las soluciones anteriores que escriben los datos formados en un flujo como **cout** también funcionan con flujos de cadena. Una vez que ha construido la cadena formada, puede desplegarla usando cualquier mecanismo que elija. En esta solución se muestra el proceso.

Paso a paso

Una manera de formar datos en una cadena requiere los siguientes pasos:

1. Cree un **ostringstream**.
2. Establezca las marcas de formato, precisión, ancho y carácter de relleno, de acuerdo con lo necesario.
3. Dé salida a los datos al flujo de cadena.
4. Para obtener la cadena formada, llame a **str()**.

Análisis

Los flujos de cadena, incluido **ostringstream**, se describieron en el capítulo 5. Consulte *Use los flujos de cadena* para conocer más detalles sobre la creación y el uso de un flujo de cadena.

Las marcas de formato, precisión, ancho y carácter de relleno se establecen en el flujo de cadena de la misma manera que en cualquier otro flujo de C++. Por ejemplo, puede utilizar la función **setf()** para establecer las marcas de formato. Use **width()**, **precision()** y **fill()** para establecer el ancho, la precisión y el carácter de relleno. Como opción, puede utilizar los manipuladores de E/S para establecer estos elementos.

Para crear una cadena formada, simplemente dé salida al flujo. Cuando quiera usar la cadena formada, llame a **str()** en el flujo de cadena para obtenerla. Con ello, podrá desplegar, almacenar o usar la cadena de la manera que guste.

Ejemplo

En el siguiente ejemplo se muestra cómo crear una cadena formada mediante el uso de un flujo de cadena. Una vez que se ha construido la cadena formada, se le da salida:

```
// Usa un flujo de cadena para almacenar salida formada en una cadena.

#include <iostream>
#include <sstream>
```

```
#include <locale>
#include <iomanip>

using namespace std;

int main()
{
  locale usloc("English_US");

  ostringstream flucadsad;

  // Establece la marca showbase para que se despliegue el símbolo de moneda.
  flucadsad << showbase;

  // Establece la configuración regional y de idioma de flucadsad en US English.
  flucadsad.imbue(usloc);

  // Obtiene una faceta money_put para flucadsad.
  const money_put<char> &mon_eu =
          use_facet<money_put<char> >(flucadsad.getloc());

  // Forma un valor en dólares de EU.
  mon_eu.put(flucadsad, false, flucadsad, ' ', "5498499");

  cout << "Dinero formado para EU: ";
  cout << flucadsad.str() << "\n\n";

  // Da una nueva cadena vacía a flucadsad.
  flucadsad.str(string());

   // Ahora, construye una tabla de áreas de un círculo.
  flucadsad << setprecision(4) << showpoint << fixed << left;
  flucadsad << "Di\u00a0metro    Area\n";

  cout << "Una tabla de \u00a0reas de un c\u00a1rculo.\n";
  for(int i=1; i < 10; ++i)
    flucadsad << left << "   " << setw(6) << i << setw(8)
         << right << i*3.1416 << endl;

  // Despliega la cadena formada.
  cout << flucadsad.str();

  return 0;
}
```

Aquí se muestra la salida:

```
Dinero formado para EU: $54,984.99

Una tabla de áreas de un círculo.
Diámetro    Área
   1       3.1416
   2       6.2832
   3       9.4248
```

```
4       12.5664
5       15.7080
6       18.8496
7       21.9912
8       25.1328
9       28.2744
```

Opciones

La función heredada de C **sprintf()** ofrece otra manera de escribir salida formada a una cadena. Se describió en la sección *Opciones* de la solución *Use **printf()** para formar datos*. Debido a las posibilidades de desbordamiento del búfer, y a que los flujos de cadena ofrecen una opción más flexible, **sprintf()** no se recomienda para nuevo código. Se incluye en este libro sólo debido a que hace uso extenso del código C heredado.

Forme la fecha y hora con strftime()

Componentes clave		
Encabezados	**Clases**	**Funciones**
<ctime>		struct tm &localtime(const time_t **hora*) size_t strftime(char **cad*, size_t *tammax*, const char **fmt*, const struct tm **apt_h*)

Aunque se recomienda usar la faceta **time_put** para casi todo el formato de fecha y hora, hay una opción que puede ser útil en algunos casos: la función **strftime()**. Está definida en C y aún tiene soporte en C++. Aunque carece de parte de la flexibilidad de la faceta **time_put** (descrita en una solución anterior), puede ser útil cuando está desplegando la fecha y hora para la configuración regional y de idioma global. En esta solución se muestra el proceso.

Paso a paso

Para usar **strftime()** para formar la fecha y hora se necesitan estos pasos:

1. Obtenga un apuntador a **tm** que señale a la hora que habrá de formarse. En el caso de la hora local, este apuntador puede obtenerse al llamar a **localtime()**.
2. Cree una matriz **char** con el largo suficiente para contener la salida formada. Recuerde incluir espacio para el terminador de carácter nulo.
3. Para formar la fecha y hora, llame a **strftime()**, especificando los formatos deseados. También pasará un apuntador a la matriz de **char** del paso 2 y un apuntador a **tm** del paso 1.

Análisis

La función **strftime()** forma la fecha y hora, poniendo el resultado en una cadena terminada en carácter nulo. Requiere el encabezado **<ctime>** y tiene el siguiente prototipo:

size_t strftime(char **cad*, size_t *tammax*, const char **fmt*,
const struct tm **apt_h*)

La hora que habrá de formarse está en una estructura **tm** a la que señala *apt_h*. El formato de la fecha y hora se especifica en la cadena a la que señala *fmt*. La salida formada se pone en la cadena a la que señala *cad*. El resultado termina con un carácter nulo. Un máximo de *tammax* caracteres se colocará en *cad*. Devuelve el número de carácter que habrá de ponerse en *cad* (excluido el terminador de carácter nulo). Debe asegurarse de que *cad* señala a una matriz con el tamaño suficiente para contener la salida máxima. Por tanto, debe tener por lo menos *tammax* elementos de largo. Se devuelve cero si se necesitan más de *tammax* caracteres para contener el resultado formado.

La función **strftime()** forma la fecha y hora de acuerdo con los *especificadores de formato*. Cada especificador empieza con el signo de porcentaje (%) y es seguido por un comando de formato. Estos comandos se utilizan para especificar la manera exacta en que se representará la distinta información de fecha y hora. Cualquier otro carácter encontrado en *fmt* (la cadena de formato) se copia en *cad*, que no cambia. La fecha y hora se forman de acuerdo con la configuración regional y de idioma global, que es "C", como opción predeterminada. Los comandos de formato se muestran en la tabla 6-1. Tome en cuenta que muchos de los comandos son sensibles a mayúsculas y minúsculas.

Para comprender la manera en que funcionan los formatos de fecha y hora, trabajemos un ejemplo. Tal vez el formato de uso más común sea **%c**, que despliega la fecha y hora usando un formato estándar apropiado para la configuración regional y de idioma. Los formatos de fecha y hora estándares pueden usarse por separado al especificar **%x** (fecha) y **%X** (hora). Por ejemplo, la cadena de formato "%x %X" hace que se desplieguen la fecha y hora estándares.

Aunque los formatos estándar son útiles, puede tomar control completo usando cualquier parte de la fecha, la hora, o ambas, que desee, de varias formas. Por ejemplo, "%H:%M" despliega la hora, usando sólo horas y minutos, en un formato de 24 horas. Observe que las horas están separadas de los minutos por dos puntos. Como se explicó, se dará salida directa a cualquier carácter en la cadena de formato que no sea parte de un especificador. He aquí un formato de fecha popular: "%A, %d de %B de %Y". Despliega el día, mes y año empleando el formato de nombre largo, como en Martes, 01 de noviembre de 2008.

En **strftime()**, el parámetro *apt_t* señala a un objeto de tipo **tm** que contiene lo que se denomina la forma "desglosada" de la hora. La estructura **tm** también se define en **<ctime>**. Una manera de obtener un objeto de **tm** consiste en llamar a la función **localtime()**. Devuelve un apuntador a una estructura **tm** que contiene la hora representada como hora local. Puede obtener la hora actual al llamar a **time()**. Consulte *Forme la fecha y hora con la faceta **time_put*** para conocer información adicional sobre **tm**, **localtime()** y **time()**.

Ejemplo

En el siguiente ejemplo se muestra la función **strftime()** en acción:

```
#include <iostream>
#include <ctime>

using namespace std;
```

```
int main() {
  char cad[64];

  // Obtiene la hora actual del sistema.
  time_t t = time(NULL);

  // Muestra la cadena estándar de fecha y hora.
  strftime(cad, 64, "%c", localtime(&t));
  cout << "Formato est\u00a0ndar: " << cad << endl;

  // Muestra una cadena de fecha y hora personalizada.
  strftime(cad, 64, "%A, %B %d %Y %I:%M %p", localtime(&t));
  cout << "Formato personalizado: " << cad << endl;

  return 0;
}
```

Comando	**Reemplazado por**
%a	Nombre abreviado del día de la semana.
%A	Nombre completo del día de la semana.
%b	Nombre abreviado del mes.
%B	Nombre completo del mes.
%c	Cadena de fecha y hora estándar.
%d	Día del mes, como decimal (1-31).
%H	Hora (0-23).
%I	Hora (1-12).
%j	Día del año, como decimal (1-366).
%m	Mes, como decimal (1-12)
%M	Minuto, como decimal (0-59)
%p	Equivalente de configuración regional y de idioma de a.m. y p.m.
%S	Segundo, como decimal (0-61).
%U	Semana del año; el domingo es el primer día (0-53).
%w	Día de la semana, como decimal (0-6; el domingo es 0).
%W	Semana del año; el lunes es el primer día (0-53).
%x	Cadena de fecha estándar.
%X	Cadena de hora estándar.
%y	Año en decimal, sin el siglo (0-99).
%Y	Año, incluido el siglo, como decimal.
%Z	Nombre de la zona horaria.
%%	El signo de porcentaje.

TABLA 6-1 Los especificadores de formato de **strftime()**.

Aquí se muestra la salida:

```
Formato estándar: 11/24/08 13:28:59
Formato personalizado: Monday, November 24 2008 01:28 PM
```

Opciones

Algunos compiladores dan soporte a modificadores de comandos de formato de fecha y hora, pero son dependientes de la implementación. Por ejemplo, Microsoft Visual C++ le permite modificar un comando con **#**. El efecto preciso varía entre comandos. Por ejemplo, **%#c** hace que la cadena de fecha y hora estándar se despliegue en su forma larga, con los nombres de los días de la semana y el mes escritos. Necesitará revisar la documentación de su compilador para conocer los modificadores que se aplican a su entorno de desarrollo.

La función **strftime()** usa la configuración regional y de idioma global definida por C para determinar los formatos de fecha y hora. Puede cambiar esta configuración al llamar a la función de C **setlocale()**, que se muestra a continuación:

char *setlocale *que*, const char **loc*)

La función **setlocale()** trata de usar la cadena especificada por *loc* para establecer los parámetros de configuración regional y de idioma como se especifica en *que*. Las cadenas de configuración son dependientes de la implementación. Consulte la documentación de su compilador para conocer las cadenas de localización a las que da soporte. Si *loc* es null, **setlocale()** devuelve un apuntador a la cadena de localización actual. Al momento de la llamada, *que* debe ser una de las siguientes macros:

LC_ALL	LC_COLLATE	LC_CTYPE
LC_MONETARY	LC_NUMERIC	LC_TIME

LC_ALL alude a todas las categorías de localización. **LC_COLLATE** afecta a las funciones de intercalación, como **strcoll()**. **LC_CTYPE** modifica la manera en que actúan las funciones de caracteres. **LC_MONETARY** determina el formato monetario. **LC_NUMERIC** determina el formato numérico. **LC_TIME** determina el comportamiento de la función **strftime()**. La función **setlocale()** devuelve un apuntador a una cadena asociada con el parámetro *que*. Para usar **setlocale()**, debe incluir **<clocale>**.

En el siguiente programa se vuelve a trabajar el ejemplo, de modo que la fecha y la hora se desplieguen en forma compatible con Spanish_Spain. (La cadena de configuración regional y de idioma es compatible con Visual C++. Tal vez su compilador requiera una cadena diferente.)

```
#include <iostream>
#include <ctime>
#include <clocale>

using namespace std;

int main() {
  char cad[64];

  // Establece la configuración regional y de idioma en Spanish_Spain.
  setlocale(LC_ALL, "Spanish_Spain");
```

```
  // Obtiene la hora actual del sistema.
  time_t t = time(NULL);

  // Muestra la cadena estándar de fecha y hora.
  strftime(cad, 64, "%c", localtime(&t));
  cout << "Formato estándar: " << cad << endl;

  // Muestra una cadena de fecha y hora personalizada.
  strftime(cad, 64, "%A,  %B %Y %I:%M %p", localtime(&t));
  cout << "Formato personalizado: " << cad << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Formato estándar: 24/11/2008 13:56:11
Formato personalizado: lunes, 24 de noviembre de 2008 01:56
```

Observe que ahora la fecha y la hora están en español y tienen el estilo propio de este idioma*.

Aunque **strftime()** en ocasiones ofrece una opción conveniente, en casi todos los casos querrá usar **time_put** para nuevo código. La razón es que el sistema de localización de C++ está completamente integrado en los flujos de C++. Más aún, cada flujo puede tener su propia configuración regional y de idioma. La función **strftime()** usa la configuración global, que es una característica heredada del lenguaje C. El método moderno es que cada flujo tenga su propia configuración regional y de idioma.

Use printf() para formar datos

Componentes clave		
Encabezados	**Clases**	**Funciones**
<cstdio>		int printf(const char **fmt*, ...)

Aunque el uso de facetas como **num_put** y **money_put** es la manera moderna de formar datos, tal vez las facetas no sean lo primero que le viene a la mente a casi ningún programador que trabaje con C++. En cambio, tal vez lo sea la función **printf()**. Incorporada en C++ como parte de su legado de C, **printf()** es, quizás, la función para formar salida más usada, mejor comprendida y más copiada que existe. Aun programadores con poco conocimiento de C o C++ han oído de ella. También se ha agregado al lenguaje Java. Aunque las marcas de formato, las funciones y las facetas definidas por los flujos de C++, en esencia, duplican su funciones, la formación con el estilo de **printf()** aún se emplea mucho porque ofrece una manera compacta de crear casi cualquier tipo de formato numérico o de cadena. También se usa ampliamente en código heredado de C. Francamente, ningún programador puede considerarse un maestro de C++ sin saber cómo manejar **printf()**.

Antes de empezar, es necesario dejar en claro un tema importante: **printf()** es sólo parte de una familia de funciones que trabajarán, en esencia, de la misma manera. Las otras funciones descritas

***Nota del T.** Debido a que se ha establecido la configuración regional y de idioma para español, no es necesario usar secuencias de escape para caracteres especiales, como letras con acentos en el flujo afectado por esa configuración.

en esta solución son **sprintf()** y **fprintf()**. Las tres forman datos mediante el uso de especificadores de formato. La diferencia entre estas funciones es el destino de la salida formada. En el caso de **printf()**, el destino es la salida estándar, que suele ser la consola. Para **sprintf()** el destino es una cadena, y para **fprintf()**, es un archivo (como se especifica en el apuntador estilo C, no un flujo de C++). Excepto por el lugar al que se envían los datos, la información presentada en esta solución se aplica a las tres funciones.

NOTA *Casi todo el código nuevo debe usar características de C++ para formación, no **printf()**. La formación de C++ está integrada en los flujos de C++ y ofrece mejor soporte a internacionalización. Además, **printf()** forma los datos de acuerdo con la configuración regional y de idioma, no una configuración basada en flujos. Por tanto, el método de C++ es más flexible. Por último, suele ser mejor no mezclar salida a **cout** con salida de **printf()**. Como regla general, para cualquier flujo dado, debe usar E/S de C++ o C. Por tanto, si quiere usar **printf()** en un programa, no debe usar **cout** en él.*

Paso a paso

Para la formación de datos mediante **printf()** se necesitan los siguientes pasos:

1. Cree una cadena de formato que contenga los especificadores de formato deseados.
2. Pase la cadena de formato como el primer argumento de **printf()**.
3. A partir del segundo argumento de **printf()**, pase los datos que desee formar. Debe haber el mismo número de argumentos que de especificadores de formato, y deben estar en el mismo orden.

Análisis

La función **printf()** escribe salida formada al dispositivo de salida estándar, que es la consola, como opción predeterminada. Se muestra a continuación:

int printf(const char **fmt*, *lista-args*)

Forma los datos pasados en *lista-args* de acuerdo con los especificadores de formato contenidos en *fmt*. Devuelve el número de caracteres que se imprimirá en realidad. Si se devuelve un valor negativo, se indica que ha ocurrido un error.

La cadena a la que señala *fmt* consta de dos tipos de elementos. El primero está integrado por caracteres que se desplegarán tal cual. El segundo tipo contiene *especificadores de formato* que definen la manera en que se formarán los argumentos. Los especificadores de formato se muestran en la tabla 6-2. Observe que todos empiezan con un signo de porcentaje y son seguidos por un código de formato. Debe haber exactamente el mismo número de argumentos que de especificadores de formato, y se asignan unos a otros en orden. Por ejemplo, la siguiente llamada a **printf()**:

```
printf("Hola %c %s %d &s", 'a', "ustedes", 10);
```

despliega

```
Hola a ustedes 10
```

Si hay argumentos insuficientes para asignar los especificadores de formato, la salida queda sin definir. Si hay más argumentos que especificadores de formato, se descartan los sobrantes. En las siguientes secciones se describen de manera detallada los especificadores de formato.

Código	Formato
%c	Carácter.
%d	Enteros decimales con signo.
%i	Enteros decimales con signo.
%e	Notación científica (e minúscula).
%E	Notación científica (E mayúscula).
%f	Punto flotante decimal.
%g	Usa %e o %f, la que sea más corta (si usa %e, la e será minúscula).
%G	Usa %E o %f, la que sea más corta (si usa %E, la E será mayúscula).
%o	Octal sin signo.
%s	Cadena terminada en un carácter nulo.
%u	Enteros decimales sin signo.
%x	Hexadecimal sin signo (letras minúsculas).
%X	Hexadecimal sin signo (letras mayúsculas).
%p	Despliega una dirección.
%n	El argumento asociado debe ser un apuntador a un entero, en que se coloca el número de caracteres escrito hasta ahora.
%%	Imprime un signo %.

TABLA 6-2 Los especificadores de formato usados por la familia de funciones **printf()**.

Forme caracteres y cadenas

Para desplegar un carácter individual, use **%c**. Para imprimir una cadena terminada en un carácter nulo, use **%s**. No puede usar **printf()** para desplegar un objeto de **string**.

Forme enteros

Puede usar **%d** o **%i** para formar un valor entero. Estos especificadores de formato son equivalentes: ambos tienen soporte por razones históricas. Para dar salida a un **unsigned int**, utilice **%u**.

Tiene la opción de desplegar un **entero sin signo** en formato octal o hexadecimal usando **%o** y **%x**, respectivamente. Debido a que el sistema numérico hexadecimal usa de la letra A a la F para representar los números del 10 al 15, puede desplegar estas letras en mayúsculas o minúsculas. Para el primer caso, utilice el especificador de formato **%X**; para minúsculas, use **%x**.

Forme valores de punto flotante

El especificador de formato **%f** despliega un argumento **double** en formato de punto flotante. Los especificadores **%e** y **%E** indican a **printf()** que despliegue un argumento **double** en notación científica. Los números representados en esta notación toman esta forma general:

x.dddddE+/-yy

Si quiere desplegar la letra "E" en mayúsculas, use el formato **%E**; de otra manera, use **%e**. Puede usar **%f** o **%e** empleando los especificadores de formato **%g** o **%G**. Esto hace que **printf()** seleccione el especificador de formato que produzca la salida más corta. Donde sea aplicable, use **%G** si quiere que "E" aparezca en mayúsculas; de otra manera, use **%g**.

Los prefijos de tipo

Para permitir que **printf()** despliegue enteros **short** y **long**, necesitará agregar un prefijo en el especificador de tipo. Estos prefijos pueden aplicarse a especificadores de tipo **d**, **i**, **o**, **u** y **x**. El modificador **l** indica que sigue un tipo de datos largo. Por ejemplo, **%ld** significa que un **long int** va a formarse. El **h** indica un **short int**. Por tanto, **%hu** indica que los datos son del tipo **short unsigned int**.

Un modificador **L** puede ser prefijo de los especificadores de punto flotante **e**, **f** y **g**, e indica que sigue un **long double**.

Si está usando un compilador moderno que da soporte a formatos de caracteres extendidos, entonces puede usar el modificador **l** con el especificador **c** para indicar un carácter extendido de tipo **whcar_t.** También puede usar el modificador I con el especificador **s** para indicar una cadena de caracteres extendidos.

Despliegue una dirección

Para desplegar una dirección, utilice el especificador **%p**. La dirección se formará de una manera compatible con el tipo de direccionamiento usado por el entorno en ejecución.

Especificador %n

El especificador **%n** es único porque en realidad no forma datos. En cambio, hace que el número de caracteres que se ha escrito en el momento en que se encuentra **%n** se almacene en una variable de entero cuyo apuntador se especifica en la lista de argumentos. Por ejemplo, este fragmento de código despliega el número 14 después de la línea "Se trata de una prueba":

```
int i;

printf("Se trata de una prueba"%n, &i);
printf("%d", i);
```

Establezca el ancho de campo y la precisión

Los especificadores de formato pueden incluir modificadores que especifican el ancho de campo y la precisión. Un entero colocado entre el signo % y el código de formato actúa como un *especificador de ancho de campo mínimo*. Esto rellena la salida para asegurar que tenga, por lo menos, cierta longitud mínima. Si la cadena o el número es mayor que el mínimo, se imprimirá completa, aunque rebase el mínimo. El relleno predeterminado se hace con espacios. Si quiere que se rellene con 0, coloque un 0 antes del especificador de ancho de campo. Por ejemplo, **%05d** rellenará un número de menos de cinco dígitos con 0 para que tenga una longitud total de 5.

El significado exacto de *modificador de precisión* depende del especificador de formato que se está modificando. Para agregar un modificador de precisión, coloque un punto decimal, seguido por la precisión, después del especificador de ancho de campo. Para los formatos **e**, **E** y **f**, el modificador de precisión determina el número de lugares decimales que se imprimirá. Por ejemplo, **%10.4f** desplegará un número de por lo menos 10 caracteres de ancho con cuatro lugares decimales. Cuando el modificador de precisión se aplica a código de formato **g** o **G**, determina el número máximo de

dígitos significativos desplegado. Cuando se aplica a enteros, el modificador de precisión especifica el número mínimo de dígitos que se desplegará. Se agregan ceros al principio, si es necesario.

Cuando el modificador de precisión se aplica a cadenas, el número después del punto especifica la longitud de campo máxima. Por ejemplo, **%5.7s** desplegará una cadena que tendrá por lo menos cinco caracteres de largo y no será mayor de siete. Si la cadena es más larga que el ancho de campo máximo, los caracteres del final se truncarán.

Los especificadores de ancho de campo y de precisión pueden alimentarse como argumentos a **printf()**, en lugar de hacerlo como constantes. Para realizar esto, utilice * como marcador de posición. Cuando se revise la cadena de formato, **printf()** asignará cada * a un argumento en el orden en que se presenten. Por ejemplo:

```
printf("|%*.*f|", 8, 3, 98.6);
```

```
producirá la siguiente salida:
|  98.600 |
```

En este ejemplo, el primer * coincidirá con 8, el segundo con 3 y **f** con 98.6.

Justifique a la izquierda la salida

Como opción predeterminada, toda la salida se justifica a la derecha. Si el ancho de campo es mayor que los datos impresos, éstos se colocarán a la derecha del campo. Puede imponer que la información se justifique a la izquierda al colocar un signo de menos directamente después de %. Por ejemplo, **%–10.2f** justificará a la izquierda un número de punto flotante con dos lugares decimales en un campo de diez caracteres.

Las marcas #, + y espacio

Además de la marca de justificación a la izquierda que se acaba de describir, **printf()** da soporte a otras tres. Son **#**, **+** y espacio. A continuación se describe cada una de ellas.

La marca **#** tiene un significado especial cuando se usa con algún especificador de formato de **printf()**. Al anteceder a **g**, **G**, **f**, **e** o **E** con una marca **#** se asegura que el punto decimal esté presente, aunque no haya dígitos decimales. Si antecede el formato **x** o **X** con **#**, el número hexadecimal se imprimirá con un prefijo **0x**. Si antecede el formato **o** con **#**, el valor octal se imprimirá con un prefijo 0. La marca no puede aplicarse a ningún otro especificador de formato.

La marca **+** indica que un valor numérico con signo siempre debe incluir un signo, como en +10 o –5.

La marca de espacio causa que se agregue un espacio al principio de valores que no son negativos.

Ejemplo

En el siguiente programa se muestran varios ejemplos de **printf()** en acción:

```
// Demuestra printf().

#include <cstdio>
#include <cmath>

using namespace std;

int main()
{
  int x = 10;
  double val = 568.345;
```

```
    // No es necesaria una llamada a printf() para incluir
  // especificadores de formato o argumentos adicionales.
  printf("Se muestra la salida a la consola.\n");

  // Despliega valores numéricos.
  printf("Los valores de x y val: %d %f\n\n", x, val);
  printf("Los valores de x en hexadecimal con may\u00a3sculas: %X\n", x);

  printf("Mezcla datos %d en %f la cadena de formato.\n\n", 19, 234.3);

  // Especifica precisiones, anchos y marcas de signo diversos.
  printf("Se muestra val con precisiones, anchos y marcas de signo diversos: \n");
  printf("|%10.2f|%+12.4f|% 12.3f|%f|\n", val, val, val, val);
  printf("|%10.2f|%+12.4f|% 12.3f|%f|\n", -val, -val, -val, -val);
  printf("\n");

  // Despliega columnas de números, justificados a la derecha.
  printf("N\u00a3meros justificados a la derecha.\n");
  for(int i = 1; i < 11; ++i)
    printf("%2d %8.2f\n", i, sqrt(double(i)));

  printf("\n");

  // Ahora, justifica a la izquierda algunas cadenas en un campo de
  // 16 caracteres. Justifica a la derecha las cantidades.
  printf("%-16s Cantidad: %3d\n", "Martillos", 12);
  printf("%-16s Cantidad: %3d\n", "Pinzas", 6);
  printf("%-16s Cantidad: %3d\n", "Desarmadores", 19);

  return 0;
}
```

Aquí se muestra la salida:

```
Se muestra la salida a la consola.
Los valores de x y val: 10 568.345000

Los valores de x en hexadecimal con mayúsculas: A
Mezcla datos 19 en 234.300000 la cadena de formato.

Se muestra val con precisiones, anchos y marcas de signo diversos:
|    568.35|   +568.3450|     568.345|568.345000|
|   -568.35|   -568.3450|    -568.345|-568.345000|

Números justificados a la derecha.
 1     1.00
 2     1.41
 3     1.73
 4     2.00
 5     2.24
 6     2.45
 7     2.65
 8     2.83
 9     3.00
10     3.16
```

```
Martillos         Cantidad:  12
Pinzas            Cantidad:   6
Desarmadores      Cantidad:  19
```

Opciones

Por mucho, la mejor manera de aprender a usar de manera efectiva **printf()** es experimentar con ella. Aunque la amplitud de la sintaxis de su formato facilita la creación de especificadores de formato muy intimidantes, todos siguen las reglas descritas en el análisis. Desglose cada formato en sus partes y le resultará fácil comprender lo que hace.

La función **printf()** no se usa para formar fecha y hora. La función de C que lo hace es **strftime()**, descrita en la solución anterior. El método de C++ consiste en usar la faceta **time_put**, descrita en *Forma la fecha y hora con la faceta* ***time_put***.

Puede construir por anticipado una cadena que contenga salida formada al llamar a **sprintf()**. Parte de la familia **printf()** de funciones, **sprintf()** funciona igual que ésta, excepto que no usa la salida estándar (por lo general, la consola), sino que escribe los datos formados en una cadena. Se muestra a continuación:

int sprintf(char **cad*, const char **fmt*, ...)

La salida formada se pone en una matriz a la que señala *cad*. El resultado termina en un carácter nulo. Por tanto, al regresar, la matriz de caracteres a la que señala *cad* contiene una cadena terminada en un carácter nulo. Devuelve el número de caracteres copiado en realidad en *cad*. (La terminación en carácter nulo no es parte de la cuenta.) Un valor negativo devuelto indica un error.

Es importante que tenga cuidado cuando use **sprintf()**, debido a la posibilidad de complicaciones en el sistema y riesgos de seguridad. Se menciona aquí principalmente debido a su amplio uso en código C heredado. En el caso de nuevos proyectos, debe usar un flujo de cadena, como **ostringstream**, para poner datos formados en una cadena. (Consulte *Forme datos en una cadena*.) Cuando use **sprintf()**, *debe asegurarse* de que la matriz a la que señala *cad* tenga el tamaño suficiente para contener la salida que recibirá, incluido el terminador de carácter nulo. Si no se sigue esta regla se tendrá un desbordamiento de búfer, que puede llevar a una brecha de seguridad o a que el sistema deje de funcionar. En ningún caso debe usar **sprintf()** en datos no verificados, como datos ingresados por un usuario. Además, no debe usar una cadena con formato ingresada por el usuario porque tiene las mismas posibilidades de acarrear problemas.

NOTA ***sprintf()*** *presenta la posibilidad de causar una caída del sistema o de provocar brechas de seguridad. No se recomienda su uso en código nuevo. Muchos compiladores proporcionan versiones no estándar de* ***sprintf()****, a menudo llamadas algo así como* ***snprintf()****, que le permiten especificar el número máximo de caracteres que se copiarán en la cadena. Si está manteniendo código C heredado, se recomienda que use ese tipo de función para trarar de evitar problemas.*

Puede enviar salida formada a un archivo al usar **fprintf()**. Se muestra a continuación:

int fprintf(FILE **aa*, const char **fmt*, ...)

Funciona igual que **printf()**, excepto que los datos formados se escriben en el archivo al que señala *aa*. El valor devuelto es el número de caracteres al que se da salida en realidad. Si ocurre un error, se devuelve un número negativo. Debido a que **fprintf()** usa el sistema de E/S de C, que se basa en apuntadores a archivos en lugar de objetos de flujo, normalmente no lo utilizará en programas de C++. Se usa ampliamente, por supuesto, en código heredado de C.

CAPÍTULO 7

Popurrí

Uno de los problemas con la escritura de un libro de programación estriba en encontrar un punto apropiado para detenerse. Hay un universo casi ilimitado de temas entre los cuales elegir, y cualquier cantidad de ellos podría merecer su inclusión. Es difícil encontrar dónde trazar la línea. Por supuesto, todos los libros deben terminar. Por tanto, siempre es necesario un punto final, sea fácil encontrarlo o no. Este libro no es la excepción.

En éste, el capítulo final del libro, el autor ha decidido concluir con una variedad de soluciones que abarcan diversos temas. Éstas representan técnicas que se desean cubrir en el libro; sin embargo, por una razón u otra, un capítulo completo no era apropiado para ninguna de ellas. Por ejemplo, se quería mostrar cómo sobrecargar los operadores de caso especial de C++, como **[]**, **–>**, **new** y **delete**, etc. Aunque varias están dedicadas a sobrecargar estos operadores, no son suficientes para un capítulo. También se desea incluir soluciones que atienden a alguna pregunta común pero aislada tipo "¿Cómo hacer?"; por ejemplo, cómo crear un constructor de copia, implementar una función de conversión o usar un ID de tipo en tiempo de ejecución. Todos son temas importantes, pero ninguno es lo suficientemente importante para merecer un capítulo propio. A pesar de la naturaleza de rango amplio de las soluciones de este capítulo, todos tienen dos cosas en común:

1. Responden una pregunta frecuente.
2. Son aplicables a un amplio rango de programadores.

Más aún, todas describen conceptos clave que puede adaptar y mejorar fácilmente.

He aquí las soluciones contenidas en este capítulo:

- Técnicas básicas de sobrecarga de operadores
- Sobrecargue el operador de llamada a función ()
- Sobrecargue el operador de subíndice []
- Sobrecargue el operador –>
- Sobrecargue **new** y **delete**
- Sobrecargue los operadores de aumento y disminución
- Cree una función de conversión
- Cree un constructor de copia
- Determine un tipo de objeto en tiempo de ejecución

- Use números complejos
- Use **auto_ptr**
- Cree un constructor explícito

Técnicas básicas de sobrecarga de operadores

Componentes clave		
Encabezados	**Clases**	**Funciones**
		tipo-ret operator#(*lista-param*)

En C++, los operadores pueden sobrecargarse en relación con una clase, incluidas las clases personalizadas. Esto le permite definir lo que una operación específica, como **+** o **/**, significa para un objeto de la clase. También permite que estos objetos sean usados en expresiones, de la misma manera en que se utilizan para usar tipos integrados. Recuerde que cuando define una clase, está creando un nuevo tipo de datos. Mediante la sobrecarga de operadores, puede integrar de manera transparente este nuevo tipo de datos en su entorno de programación. Esta *extensibilidad de tipo* es una de las características más importantes y poderosas de C++ porque le permite expandir el sistema de tipos de C++ para cubrir sus necesidades.

La sobrecarga de operadores será un territorio familiar para la mayoría de los lectores porque es una habilidad básica de C++ y casi todos los programadores saben cómo sobrecargar los operadores de uso más común. Por esto, la solución de sobrecarga de operadores de este capítulo se concentra en estos operadores especializados: aumento y reducción, **()**, **[]**, **–>**, **new** y **delete**. A muchos programadores les parecen estos operadores confusos cuando se trata de sobrecarga, y son la fuente de muchas preguntas tipo "¿Cómo hacer?". Sin embargo, para proporcionar la información completa, en esta solución se presenta una breve revisión general de las técnicas básicas usadas para sobrecargar un operador. Esta revisión general es suficiente para los propósitos de este capítulo, pero no es un sustitutivo de un examen a profundidad del tema.

NOTA *Para una revisión a profundidad de la sobrecarga de operadores, se recomienda el libro* C++: The Complete Reference, *de Herb Schildt.*

Paso a paso

Para sobrecargar un operador como una función miembro de una clase se requieren estos pasos:

1. Agregue una función **operator** a la clase, especificando el operador que quiera sobrecargar.
2. En el caso de operadores binarios, la función **operator** tendrá un parámetro, que recibirá el operando del lado derecho. El operador del lado izquierdo se pasará mediante **this**.
3. En el caso de operadores unarios, la función **operator** no tendrá parámetros. Su único operando se pasa mediante **this**.

4. En el cuerpo de la función, realice la operación.
5. Regrese el resultado de la operación.

Para sobrecargar un operador como una función que no es miembro se requieren estos pasos:

1. Cree una función **operator** que no sea miembro, que especifique el operador que quiera sobrecargar.
2. En el caso de operadores binarios, la función **operator** tendrá dos parámetros. El primer parámetro recibe el operando del lado izquierdo, y el segundo recibe el operando del lado derecho. Por lo menos uno de los operandos debe ser un objeto de la clase sobre la que se está actuando o una referencia a éste.
3. En el caso de operadores unarios, la función **operator** tendrá un parámetro, que debe ser un objeto de una referencia a la clase sobre la que se está actuando. Este parámetro es el operando.
4. En el cuerpo de la función, realice la operación.
5. Regrese el resultado de la operación.

Análisis

Cuando sobrecarga un operador, define el significado de ese operador para una clase particular. Por ejemplo, una clase que define una lista vinculada podría usar el operador **+** para agregar un objeto a la lista. Una clase que implementa una pila podría usar el **+** para incluir un objeto en la pila. Otra clase podría usar el operador **+** de una manera completamente diferente. Cuando se sobrecarga un operador, ninguno de sus significados originales se pierde. Simplemente se define una nueva operación, relacionada con una clase específica. La sobrecarga de **+** para manejar una lista vinculada, por ejemplo, no causa que cambie su significado relacionado con los enteros (es decir, la suma).

Para sobrecargar un operador, debe definir lo que significa la operación en relación con la clase a la que se aplica. Como regla general, puede usar funciones miembro o no miembro. (Las excepciones a esta regla son las funciones de operador para **=**, **()**, **[]** y **–>**, que debe implementarse por una función miembro no estática.) Aunque son similares, hay algunas diferencias entre los dos métodos.

Para crear una función de operador, utilice la palabra clave **operator**. Su forma general es:

```
tipo-ret operator#(lista-param)
{
  // operaciones
}
```

Aquí, el operador que está sobrecargando se sustituye con #, y *tipo-ret* es el tipo de valor devuelto por la operación especificada. Aunque puede ser del tipo que elija, el valor devuelto es a menudo del mismo tipo que la clase para la que se está sobrecargando el operador. Esta correlación facilita el uso del operador sobrecargado en expresiones compuestas. Las excepciones son los operadores lógicos y relacionales, que suele regresar un valor **bool**.

La naturaleza precisa de *lista-param* depende del tipo de operador que se está sobrecargando y si está implementado como una función miembro o no miembro. En el caso de una función **operator** unaria miembro, *lista-param* estará vacía y el operando se pasa a través del apuntador **this**. Para una función **operator** binaria miembro, *lista-param* tendrá un parámetro, que recibe el operando del lado derecho. El operando del lado izquierdo se pasa mediante **this**. En cualquier caso, el objeto que invoca la función de operador es el pasado mediante el apuntador **this**.

En el caso de funciones **operator** que no son miembros, todos los argumentos se pasan explícitamente. Por tanto, una función **operator** unaria que no es miembro tendrá un parámetro, cuyo tipo debe ser una clase, referencia a clase, enumeración o referencia a enumeración. Este parámetro recibe el operando. Una función **operator** binaria que no es miembro tendrá dos parámetros, de los cuales el tipo de por lo menos uno debe ser una clase, referencia a clase, enumeración o referencia a enumeración. El primer parámetro recibe el operando del lado izquierdo y el segundo recibe el operando del lado derecho. Observe que una función de operador que no es miembro puede sobrecargarse en relación con un tipo de enumeración, pero esto no es común. Por lo general, los operadores están sobrecargados en relación con un tipo de clase, y ése es el eje de esta solución.

Debido a las diferencias entre las funciones de operador miembros y no miembros, cada una se describe por separado.

Funciones de operador miembro

Cuando se define una función **operator** que actúa sobre objetos de una clase que haya creado, por lo general usará una función miembro. La razón es simple: siendo miembro de una clase, la función tiene acceso directo a todos los miembros de clase. También tiene un apuntador **this**. Esto facilita que el operador actúe sobre un operando, y posiblemente lo modifique.

La mejor manera de comprender cómo usar una función miembro para sobrecargar un operador consiste en trabajar con algunos ejemplos. Suponga una clase llamada **tres_d** que encapsula coordinadas tridimensionales, como se muestra aquí:

```
class tres_d {
  int x, y, z; // Coordenadas 3-D
public:
  tres_d() { x = y = z = 0; }
  tres_d(int i, int j, int k) { x = i; y = j; z = k; }

  //...
};
```

Puede definir la operación **+** para objetos de **tres_d** al agregar una función **operator+()** a la clase. Para ello, primero agregue su prototipo a la clase **tres_d**:

```
tres_d operator+(tres_d op_der);
```

Luego, implemente la función. He aquí una manera:

```
// Sobrecarga + para objetos de tipo tres_d.
tres_d tres_d::operator+(tres_d op_der)
{
  tres_d temp;

  temp.x = x + op_der.x;
  temp.y = y + op_der.y;
  temp.z = z + op_der.z;

  return temp;
}
```

Esta función agrega las coordenadas de dos operandos de **tres_d** y devuelve un objeto que contiene el resultado. Recuerde que en una función de operador miembro, el operando del lado izquierdo invoca a la función de operador y se pasa implícitamente mediante **this**. El operando del lado derecho se pasa de manera explícita como un argumento a la función. Por tanto, suponiendo que **objA** y **obj** son objetos de **tres_d**, en la siguiente expresión

```
objA + obj B
```

objA se pasa mediante **this** y **obj s**e pasa en **op_der**.

En la implementación de **operator+()** que se acaba de mostrar, observe que ningún operando se modifica. Esto es para seguir con la semántica normal del operador **+**. Por ejemplo, en la expresión 10 + 12, ni el 10 ni el 12 se modifican. Aunque no hay una regla para imponerlo, en general es mejor hacer que su operador sobrecargado trabaje de la manera esperada.

Por supuesto, hay algunos operadores, como asignación o aumento, en que un operando *se* modifica con la operación. En este caso, necesitará modificar un operando para que su función **operator** refleje el significado normal del operador. Por ejemplo, suponiendo una vez más la clase **tres_d**, hay una manera de implementar la asignación:

```
// Sobrecarga asignación para tres_d.
tres_d tres_d::operator=(tres_d op_der)
{
  x = op_der.x;
  y = op_der.y;
  z = op_der.z;

  return *this;
}
```

Aquí, los valores de coordenadas del operando del lado derecho (pasado en **op_der**) se asignan al operando del lado izquierdo (pasado mediante **this**). Por tanto, el objeto que invoca se cambia para reflejar el valor que se está asignando. Una vez más, esto está de acuerdo con el significado esperado de **=**.

Dadas las dos funciones **operator** que se acaban de describir y suponiendo los objetos de **tres_d** llamados **objA**, **objB** y **objC**, la siguiente instrucción es válida:

```
objC = objA + objB;
```

En primer lugar, la suma se realiza con **operator+()**; **objA** se pasa mediante **this** y **objB** se pasa a través de **op_der**. El resultado se vuelve el operando del lado derecho pasado a **operator=()**, y **objC** se pasa mediante **this**. Para una revisión más completa, **objC** contendrá la suma de **objA** y **objB**, y **objA** y **objB** quedarán sin cambio.

La versión anterior de **operator+()** sumó un objeto de **tres_d** a otro, pero puede sobrecargar **operator+()** para que agregue algún otro tipo de valor. Por ejemplo, esta versión de **operator+()** suma un entero a cada coordenada:

```
// Sobrecarga + para sumar un entero a un objeto de tres_d.
tres_d tres_d::operator+(int op_der)
{
  tres_d temp;

  temp. x = x + op_der;
```

```
  temp. y = y + op_der;
  temp. z = z + op_der;

  return temp;
}
```

Una vez que se ha definido esta versión de **operator+()**, puede usar una expresión como:

```
objA + 10
```

Esto causa que 10 se sume a cada coordenada. Comprenda que la versión anterior de **operator+()**, que suma dos objetos de **tres_d**, aún está disponible. Es sólo que la definición de **+** relacionado con **tres_d** se ha expandido para manejar la suma de enteros.

En el caso de una función de operador miembro unario, el único operando se pasa mediante **this**. Por ejemplo, he aquí la versión de **operator–()**, que niega la coordenada y devuelve el resultado:

```
// Sobrecarga - para tres_d.
tres_d tres_d::operator-()
{
  tres_d temp;

  temp.x = -x;
  temp.y = -y;
  temp.z = -z;

  return temp;
}
```

Es posible crear una forma unaria y binaria de algunos operadores, como **+** y **–**. Simplemente sobrecargue la función de operador de acuerdo con lo necesario. En el caso de funciones miembro, la forma binaria tendrá un parámetro; la forma unaria no tendrá ninguno.

Todas las funciones **operator** anteriores regresarán un objeto de tipo **tres_d**, que es la clase para la que están definidos. Así suele suceder siempre, excepto cuando sobrecarga los operadores lógicos o relacionales. Esas funciones **operator** por lo general regresarán un resultado **bool**, que indica el éxito o la falla de la operación. Por ejemplo, he aquí una manera de implementar el operador **==** para **tres_d**:

```
// Sobrecarga == para un objeto de tres_d.
bool tres_d::operator==(tres_d op_der)
{
  if( (x == op_der.x) && (y == op_der.y) && (z == op_der.z) )
    return true;

  return false;
}
```

Compara si un objeto de **tres_d** es menor que otro. Todos los valores del objeto que invoca deben ser menores que los del operando que se encuentra a la derecha para que esta función devuelva true.

Funciones de operador que no son miembros

Como se mencionó al principio de este análisis, una función de operador binario que no es miembro pasa sus operandos explícitamente, mediante sus parámetros. (Recuerde que las funciones que no son miembros no tienen apuntadores **this** porque no se invocan en un objeto.) Una función de operador binario que no es miembro tiene dos parámetros, y el operando de la izquierda se pasa al primer parámetro y el de la derecha al segundo. Una función de operador unario que no es miembro pasa su operando mediante su parámetro. De otra manera, las funciones de operador que no son miembro trabajan de modo parecido a las que sí lo son.

Aunque a menudo usará funciones miembro cuando sobrecarga operadores, hay ocasiones en que necesitará usar funciones de operador que no son miembro. Un caso es cuando quiere permitir el uso de un tipo integrado (como **int** o **char** *) en el lado izquierdo de un operador binario. Para comprender por qué, recuerde que el objeto que invoca una función de operador miembro se pasa en **this**. En el caso de un operador binario, siempre es el objeto de la izquierda el que invoca a la función. Esto es correcto, siempre y cuando el objeto de la izquierda defina la operación especificada. Por ejemplo, suponiendo un objeto de **tres_d** llamado **objA** y la función **operator+()** mostrada antes, la siguiente es una expresión perfectamente válida:

```
objA + 10; // funcionará
```

Debido a que **objA** está en el lado izquierdo del operador **+**, invoca a la función miembro sobrecargada **operator+(int)**, que suma 10 a **objA**. Sin embargo, esta instrucción no es correcta:

```
10 + Ob; // no funcionará
```

El problema es que el objeto a la izquierda del operador **+** es un entero, un tipo integrado para el que no está definida ninguna operación relacionada con un entero y un objeto de tipo **tres_d**.

La solución a este problema está en sobrecargar el **+** por segunda ocasión, empleando una función de operador que no es miembro para manejar el caso en que el entero está a la izquierda. Por tanto, la función de operador miembro maneja *objeto* **+** *entero*, y la función de operación no miembro maneja *entero* + *objeto*. Para dar a esta función acceso a los miembros de la clase, declárela como **friend**. He aquí la manera en que una versión que no es miembro de **operator+()** puede implementarse para manejar *entero* + *objeto* para la clase **tres_d**:

```
// Sobrecarga operator+() para int + obj.
// Se trata de una función que no es miembro.
tres_d operator+(int op_izq, tres_d op_der) {
  tres_d temp;

  temp.x = op_izq + op_der.x;
  temp.y = op_izq + op_der.y;
  temp.z = op_izq + op_der.z;

  return temp;
}
```

Ahora la instrucción

```
10 + Ob; // ahora es correcta
```

es legal.

Otra ocasión en que una función de operador que no es miembro resulta útil es cuando se crea un insertador o extractor personalizado. Como se explicó en el capítulo 5, **<<** se sobrecarga para que dé salida a datos (los inserte) en un flujo, y **>>** se sobrecarga para que dé entrada a datos (los extraiga) de un flujo. Estas funciones no deben ser miembros porque cada una toma un objeto de flujo como operando del lado izquierdo. El operando del lado derecho es un objeto al que se dará salida o uno que se recibirá entrada. Consulte *Cree insertadores y extractores* en el capítulo 5, para conocer más detalles.

Un último tema: no todos los operadores pueden implementarse mediante funciones que no son miembro. Por ejemplo, el operador de asignación debe ser un miembro de su clase. También lo deben ser los operadores **()**, **[]** y **–>**.

Ejemplo

En el siguiente ejemplo se pone en acción el análisis anterior, utilizando todas las piezas y demostrando los operadores.

```
// Demuestra los fundamentos de la sobrecarga de operadores usando
// la clase tres_d.
//
// Este ejemplo usa funciones miembro para sobrecargar los operadores
// binarios +, -, = y ==. También usa una función miembro para
// sobrecargar el - unario. Observe que el + se sobrecarga para
// tres_d + tres_d, y para tres_d + int.
//
// Las funciones que no son miembros se usan para crear un insertador
// predeterminado para objetos de tres_d, y para sobrecargar + para
// int + tres_d.

#include <iostream>

using namespace std;

// Una clase que encapsula coordenadas tridimensionales.
class tres_d {
  int x, y, z; // Coordenadas 3-D
public:
  tres_d() { x = y = z = 0; }
  tres_d(int i, int j, int k) { x = i; y = j; z = k; }

  // Suma dos objetos de tres_d.
  tres_d operator+(tres_d op_der);

  // Suma un entero a un objeto de tres_d.
  tres_d operator+(int op_der);

  // Resta dos objetos de tres_d.
  tres_d operator-(tres_d op_der);

  // Sobrecarga la asignación.
  tres_d operator=(tres_d op_der);

  // Sobrecarga ==.
  bool operator==(tres_d op_der);
```

```
   // Sobrecarga para operación unaria.
  tres_d operator-();

  // Hace que el insertador sobrecargado sea un amigo.
  friend ostream &operator<<(ostream &flujo, tres_d op);

  // Hace que el + sobrecargado sea un amigo.
  friend tres_d operator+(int op_izq, tres_d op_der);
};

// Sobrecarga el + binario para que se agreguen las coordenadas
// correspondientes.
tres_d tres_d::operator+(tres_d op_der)
{
  tres_d temp;

  temp.x = x + op_der.x;
  temp.y = y + op_der.y;
  temp.z = z + op_der.z;

  return temp;
}

// Sobrecarga el + binario para que pueda sumarse un entero a
// un objeto de tres_d.
tres_d tres_d::operator+(int op_der)
{
  tres_d temp;

  temp.x = x + op_der;
  temp.y = y + op_der;
  temp.z = z + op_der;

  return temp;
}

// Sobrecarga el - binario para que se resten las coordenadas
// correspondientes.
  tres_d tres_d::operator-(tres_d op_der)
{
  tres_d temp;

  temp.x = x - op_der.x;
  temp.y = y - op_der.y;
  temp.z = z - op_der.z;

  return temp;
}

// Sobrecarga el - unario, para que niegue las coordenadas.
tres_d tres_d::operator-()
{
  tres_d temp;
```

```
  temp.x = -x;
  temp.y = -y;
  temp.z = -z;

  return temp;
}

// Sobrecarga asignación para tres_d.
tres_d tres_d::operator=(tres_d op_der)
{
  x = op_der.x;
  y = op_der.y;
  z = op_der.z;

  return *this;
}

// Sobrecarga == para un objeto de tres_d. Compara cada
// coordenada. Todos los valores del objeto que invoca
// deben ser iguales a los del operando de la derecha de
// esta función para que regrese true.
bool tres_d::operator==(tres_d op_der)
{
  if( (x == op_der.x) && (y == op_der.y) && (z == op_der.z) )
    return true;

  return false;
}

// Éstas son funciones de operador que no son miembros.
//
// Sobrecarga << como un insertador personalizado para objetos de tres_d.
ostream &operator<<(ostream &flujo, tres_d op) {
  flujo << op.x << ", " << op.y << ", " << op.z << endl;

  return flujo;
}

// Sobrecarga + para int + obj.
tres_d operator+(int op_izq, tres_d op_der) {
  tres_d temp;

  temp.x = op_izq + op_der.x;
  temp.y = op_izq + op_der.y;
  temp.z = op_izq + op_der.z;

  return temp;
}

int main()
{
  tres_d objA(1, 2, 3), objB(10, 10, 10), objC;

  cout << "Esto es objA: " << objA;
```

```
  cout << "Esto es objB: " << objB;

  // Obtiene la negación de objA.
  objC = -objA;
  cout << "Esto es -objA: " << objC;

  // Suma objA a objB.
  objC = objA + objB;
  cout << "objA + objB: " << objC;

  // Resta objB a objA.
  objC = objA - objB;
  cout << "objA - objB: " << objC;

  // Suma obj + int.
  objC = objA + 10;
  cout << "objA + 10: " << objC;

  // Suma int + obj.
  objC = 100 + objA;
  cout << "100 + objA: " << objC;

  // Compara dos objetos.
  if(objA == objB) cout << "objA es igual que objB.\n";
  else cout << "objA no es igual a objB.\n";

  return 0;
}
```

Aquí se muestra la salida:

```
Esto es objA: 1, 2, 3
Esto es objB: 10, 10, 10
Esto es -objA: -1, -2, -3
objA + objB: 11, 12, 13
objA - objB: -9, -8, -7
objA + 10: 11, 12, 13
100 + objA: 101, 102, 103
objA no es igual a objB.
```

Opciones

Aunque en los ejemplos anteriores se ha pasado operandos **tres_d** por valor, en muchos casos también puede pasar un operando por referencia. Por ejemplo, he aquí **operator==()** cambiado para que el operando del lado derecho se pase por referencia:

```
bool tres_d::operator==(tres_d &op_der)
{
  if( (x == op_der.x) && (y == op_der.y) && (z == op_der.z) )
    return true;

  return false;
}
```

A menudo, el uso de una referencia puede aumentar el rendimiento de su programa, porque suele ser más rápido pasar una referencia en lugar de un objeto completo. Sin embargo, tenga cuidado. En el caso de objetos muy pequeños, el paso por valor puede ser más rápido.

Un lugar donde un parámetro de referencia es valioso es cuando un operando debe modificarse con el operador. Uno de estos casos ocurre cuando una función **operator** que no es miembro se usa para implementar una operación de aumento o reducción. Consulte *Sobrecargue los operadores de aumento y reducción* para conocer más información.

C++ tiene varios operadores de caso especiales, como el operador de llamada a función **()** o el de subíndice **[]**. Estos operadores también pueden sobrecargarse, pero las técnicas para ello están individualizadas para cada operador. Estos operadores especiales de caso son el tema de varias de las siguientes soluciones.

Hay algunas restricciones que se aplican a la sobrecarga del operador:

1. No puede modificar la precedencia de algún operador.
2. No puede modificar el número de operandos necesarios para un operador, aunque puede elegir que se ignore un operando.
3. Con excepción del operador de llamada a función **()**, las funciones de operador no pueden tener argumentos predeterminados.
4. No es posible sobrecargar los siguientes operadores:
   . :: .* ?

Desde el punto de vista técnico, tiene la libertad de realizar cualquier actividad dentro de una función de operador y no es necesario que mantenga alguna relación con el significado normal del operador. Sin embargo, cuando se aparta considerablemente del significado normal de un operador, corre el riesgo de desestructurar peligrosamente su programa. Por ejemplo, cuando alguien que lee su programa ve una instrucción como **Ob1+Ob2**, espera algo parecido a la suma, o por lo menos relacionado con ella. La implementación de **+** para que actúe más como el operador **||**, por ejemplo, es inherentemente confusa. Por tanto, antes de desacoplar un operador sobrecargado de su significado normal, asegúrese de que tiene razones suficientes para hacerlo.

Un buen ejemplo en que el desacoplamiento es correcto se encuentra en la manera en que C++ sobrecarga los operadores **<<** y **>>** para E/S. Aunque las operaciones de E/S no tienen relación con el desplazamiento de bits, estos operadores proporcionan una "pista" visual de su significado, y el desacoplamiento funciona. He aquí otro buen ejemplo de desacoplamiento: una clase de pila podría sobrecargar el **+** para poner un objeto en una pila. Aunque este uso difiere de la suma, aún es intuitivamente compatible con la suma porque "añade" un objeto a la pila.

Con excepción del operador **=**, las funciones de operador son heredadas por las clases derivadas. Sin embargo, una clase derivada tiene la libertad de sobrecargar cualquier operador que elija (incluidas las sobrecargas por una clase de base).

Sobrecargue el operador de llamada a función ()

Componentes clave		
Encabezados	**Clases**	**Funciones**
		tipo-ret operator()(*lista-param*)

Uno de los operadores más poderosos que puede sobrecargar es **()**, el operador de llamada a función. También puede ser uno de los más confusos, sobre todo para los recién llegados. El operador de llamada a función le permite definir una operación en un objeto que no puede realizarse al sobrecargar cualquier otro operador. Por ejemplo, tal vez quiera definir una operación que toma más de dos operadores. O quizá desee definir una operación que no tiene una analogía obvia con cualquiera de los operadores normales. En este caso, el operador de llamada a función ofrece una solución elegante. En esta solución se muestra el proceso.

Paso a paso

Para sobrecargar el operador de llamada a función **()** se necesitan estos pasos:

1. El operador de llamada a función debe ser un miembro no estático de la clase para la que está definido. No puede ser una función que no sea miembro. Por tanto, agregue **operator()** como miembro a la clase en que estará operando.
2. Dentro de **operator()**, realice las acciones deseadas.
3. Al terminar, haga que **operator()** devuelva el resultado.

Análisis

Cuando sobrecarga el operador de llamada a función **()**, no está creando, en sí, una nueva manera de llamar a una función. En cambio, está creando una función **operator** que puede pasarse en un número arbitrario de operandos mediante el uso de la sintaxis de llamada a función. El operador de llamada a función debe implementarse como una función miembro no estática de una clase. La forma general del operador se muestra aquí:

```
tipo-ret operator#(lista-param) {
  // realiza la operación basada en los argumentos
  // y devuelve el resultado.
}
```

El operador de llamada a función se invoca en un objeto de su clase. El objeto que invoca se pasa mediante **this**, y los argumentos se pasan a sus parámetros. Si no se necesitan argumentos, entonces no es necesario que se especifiquen parámetros. La función devuelve el resultado de la operación.

Trabajemos con un ejemplo. Suponiendo la clase **tres_d** de la solución anterior, el siguiente operador de llamada a función devuelve un objeto de **tres_d** que representa un punto cuyas coordenadas son puntos medios entre el objeto que invoca y su argumento de **tres_d**.

```
// Sobrecarga la llamada a función. Toma un objeto de tres_d como
// un parámetro. Esta función devuelve un objeto de tres_d cuyas
// coordenadas son los puntos medios entre el objeto que invoca y obj.
tres_d tres_d::operator()(tres_d obj)
{
  tres_d temp;

  temp.x = (x + obj.x) / 2;
  temp.y = (y + obj.y) / 2;
  temp.z = (z + obj.z) / 2;

  return temp;
}
```

Dados tres objetos de **tres_d** llamados **objA**, **objB** y **objC**, lo siguiente llama a **operator()** en **objA**, pasando en **objB**:

```
objC = objA(objB);
```

Aquí, **objA(objB)** se traduce en esta llamada a la función **operator()**:

```
objA.operator()(objB)
```

El resultado se devuelve y almacena en **objC**.

Antes de seguir adelante, revisemos los elementos clave. En primer lugar, cuando sobrecarga el operador **()**, define los parámetros que quiere pasar a esa función. Cuando usa el operador **()** en su programa, los argumentos que especifique se copian en esos parámetros. El objeto que genera la llamada (**objA** en el ejemplo anterior) se señala mediante el apuntador **this**.

Puede sobrecargar **operator()** para permitir diferentes tipos o cantidades de argumentos, o ambos. Por ejemplo, he aquí una versión de **operator()** para **tres_d** que toma tres argumentos **int**. Agrega los valores de esos argumentos a las coordenadas del objeto que invoca y devuelve el resultado.

```
// Sobrecarga la llamada a función. Toma tres int como parámetros.
// Esta versión suma los argumentos a las coordenadas.
tres_d tres_d::operator()(int a, int b, int c)
{
  tres_d temp;

  temp.x = x + a;
  temp.y = y + b;
  temp.z = z + c;

  return temp;
}
```

Esta función permite el siguiente tipo de instrucción:

```
objC = objA(1, 2, 3);
```

Aquí, los valores 1, 2 y 3 se agregan a los campos **x**, **y** y **z** de **objA**, y el resultado se devuelve y almacena en **objC**.

Un tema adicional: también puede sobrecargar **operator()** para que su lista de parámetros esté vacía. En este caso, no se pasan argumentos a la función cuando se le llama.

Ejemplo

En el siguiente ejemplo se unen las piezas descritas en el análisis.

```
// Demuestra el operador de llamada a función.

#include <iostream>

using namespace std;

// Una clase que encapsula coordenadas tridimensionales.
class tres_d {
  int x, y, z; // Coordenadas 3-D
public:
  tres_d() { x = y = z = 0; }
  tres_d(int i, int j, int k) { x = i; y = j; z = k; }

  // Crea dos funciones de operador de llamada a función.
  tres_d operator()(tres_d obj);
  tres_d operator()(int a, int b, int c);

  // Hace que el insertador sobrecargado sea un amigo.
  friend ostream &operator<<(ostream &flujo, tres_d op);
};

// Sobrecarga la llamada a función. Toma un objeto de tres_d como
// un parámetro. Esta función devuelve un objeto de tres_d cuyas
// coordenadas son los puntos medios entre el objeto que invoca y obj.
tres_d tres_d::operator()(tres_d obj)
{
  tres_d temp;

  temp.x = (x + obj.x) / 2;
  temp.y = (y + obj.y) / 2;
  temp.z = (z + obj.z) / 2;

  return temp;
}

// Sobrecarga la llamada a función. Toma tres int como parámetros.
// Esta versión suma los argumentos a las coordenadas.
tres_d tres_d::operator()(int a, int b, int c)
{
  tres_d temp;

  temp.x = x + a;
  temp.y = y + b;
  temp.z = z + c;

  return temp;
}

// El insertador tres_d es una función de operador que no es miembro.
ostream &operator<<(ostream &flujo, tres_d op) {
  flujo << op.x << ", " << op.y << ", " << op.z << endl;
```

```
  return flujo;
}

int main()
{
  tres_d objA(1, 2, 3), objB(10, 10, 10), objC;

  cout << "Esto es objA: " << objA;
  cout << "Esto es objB: " << objB;

  objC = objA(objB);
  cout << "objA(objB): " << objC;

  objC = objA(10, 20, 30);
  cout << "objA(10, 20, 30): " << objC;

  // Puede usar el resultado de uno como argumento de otro.
  objC = objA(objB(100, 200, 300));
  cout << "objA(objB(100, 200, 300)): " << objC;

  return 0;
}
```

Aquí se muestra la salida:

```
Esto es objA: 1, 2, 3
Esto es objB: 10, 10, 10
objA(objB): 5, 6, 6
objA(10, 20, 30): 11, 22, 33
objA(objB(100, 200, 300)): 55, 106, 156
```

Opciones

Cuando se implementa el operador de llamada a función para una clase, puede usarse una instancia de su clase como un *objeto de función*. Estos objetos se usan exclusivamente con la STL. En el capítulo 4 se muestran varios ejemplos.

Ninguna de las dos versiones de **operator()** del ejemplo anterior modifica al objeto que invoca. En cambio, devuelven el resultado. Aunque no hay una regla que lo imponga, es preferible este método en casi todos los casos. En general, si va a modificarse un objeto, es mejor que se presente mediante un operador de asignación sobrecargado, no mediante el operador de llamada a función. En otras palabras, normalmente no debe usarse

```
objA(objB);
```

como sustituto de

```
objA = objB;
```

En general, **operator()** debe reservarse para operaciones que no se relacionan con ninguna de las otras operaciones. No es adecuado usar **operator()** como un operador "total" que sustituya a una sobrecarga del operador apropiado. Usado de manera apropiada, **operator()** es una característica poderosa. Mal usado, puede crear confusión en su código.

Sobrecargue el operador de subíndice []

Componentes clave		
Encabezados	**Clases**	**Funciones**
		tipo-ret operator[](*tipo_ind ind*)

Si se tiene un operador favorito para sobrecarga, probablemente será **[]**, el operador de subíndice. ¿Por qué? Porque permite la creación de matrices "seguras", que son aquellas en que se evita el desbordamiento de límites. Como lo sabe, C++ no realiza revisión de límites en las matrices normales. Sin embargo, al envolver una matriz en una clase y luego permitir que sólo se tenga acceso a esa matriz mediante el operador de subíndice, puede evitar el acceso desde el exterior de la matriz. También puede asegurarse de que sólo se asignen valores válidos a la matriz. Este mecanismo se emplea con gran éxito en la STL, como en las clases **vector** y **deque**.

Por supuesto, el **[]** es útil en otros contextos. Por ejemplo, una clase que encapsula una solicitud de IP podría permitir el acceso a propiedades al indizar el objeto. En esencia, cada vez que tenga una clase con elementos para los que tiene sentido la indización, el operador de subíndice ofrece un método elegante. En esta solución se muestran las técnicas básicas necesarias para implementarla.

Paso a paso

La sobrecarga del operador de subíndice **[]** requiere estos pasos:

1. El operador de subíndice debe ser un miembro no estático de la clase para la que está definido. No puede ser una función que no sea miembro. Por tanto, agregue **operator[]()** como un miembro de la clase en que estará operando.
2. Dentro de **operator[]()**, realice la acción deseada, que suele incluir el acceso a algún objeto mediante un índice.
3. Al terminar, haga que **operator[]()** devuelva el objeto (o la referencia al objeto) con base en el índice.

Análisis

El **[]** es un operador binario para los fines de la sobrecarga, y debe sobrecargarse con una función miembro no estática. Tiene la forma general:

tipo-ret operator[](*tipo_ind ind*)
{
  // Accede al elemento especificado por *ind*.
}

El subíndice se pasa en *ind*, que suele ser un **int**, pero puede ser cualquier tipo. Por ejemplo, en un contenedor asociativo, *ind* puede ser una clave. La función puede devolver cualquier tipo, pero por lo general será el tipo de elemento que se está obteniendo.

Cuando se evalúa el **[]**, el objeto del subíndice debe ser una instancia de la clase para la que está definido el operador de subíndice. Esta instancia se pasa mediante **this**. El objeto dentro de **[]** se pasa en *ind*. Por ejemplo, dado un objeto llamado **obj**, la expresión

```
obj[5]
```

se traduce en esta llamada a la función **operator[]()**:

```
obj.operator[](5)
```

En este caso, 5 se pasa en el parámetro *ind*. Un apuntador a **obj**, el objeto que generó la llamada, se pasa mediante **this**.

Puede designar la función **operator[]()** de manera tal que **[]** pueda usarse a la izquierda y a la derecha de la instrucción de asignación. Para ello, simplemente especifique el valor de devolución de **operator[]()** como una referencia. Después de hacer esto, las siguientes expresiones son válidas:

```
x = obj[4];
obj[5] = 9;
```

La sobrecarga del operador **[]** proporciona un medio para implementar la indización segura de matrices en C++. Éste es uno de sus principales usos y una de sus ventajas más importantes. Como sabe, en C++ es posible desbordar el límite de una matriz (o quedarse corto con él) en tiempo de ejecución. Sin embargo, si crea una clase que contiene la matriz y sólo permite el acceso a esa matriz mediante el operador de subíndice **[]** sobrecargado, entonces interceptará cualquier índice fuera del rango. En el siguiente ejemplo se ilustra esto.

Ejemplo

En el siguiente programa se muestra cómo sobrecargar el operador de subíndice al usarlo para crear una "matriz segura" que evite errores de límite. Se define una clase genérica llamada **matriz_segura**, que encapsula una matriz. El tipo de ésta se especifica con un parámetro de tipo de plantilla llamado **T**. La longitud de la matriz se especifica con un parámetro de plantilla sin tipo llamado **longi**. La matriz encapsulada por **matriz_segura** se denomina **matriz**. La longitud de la matriz está almacenada en una variable llamada **longitud**. Ambas son miembros privados de **matriz_segura**. Se tiene acceso a los elementos de la matriz mediante el **operator[]()** sobrecargado. Primero se confirma que el acceso de una matriz está dentro de los límites. Si es así, **operator[]()** devuelve entonces una referencia al elemento. La longitud de la matriz puede obtenerse al llamar al método **getlen()**.

```
// Sobrecarga [] para crear un tipo de matriz segura genérica.
//
// La función operator[]() revisa errores de límite de matriz
// para que se evite un rebase de límites o que se quede corto de él.
//
// Observe que en este ejemplo se usa un parámetro de plantilla sin
// tipo para especificar el tamaño de la matriz.

#include <iostream>
#include <cstdlib>

using namespace std;
```

```
// Aquí, T especifica el tipo de matriz y el parámetro sin tipo
// longi especifica la longitud de la matriz.
template <class T, int longi> class matriz_segura {

  // La matriz mz está declarada como de tipo T y de longitud longi.
  // La matriz es privada. El acceso sólo se permite con operator[]().
  // De esta manera, pueden evitarse los límites de error.
  T mz[longi];
  int longitud;

public:
  // Crea una matriz_segura de tipo T con una longitud longi.
  matriz_segura();

  // Sobrecarga el operador de subíndice, de modo que acceda a los
  // elementos de mz.
  T &operator[](int i);

  // Devuelve la longitud de la matriz.
  int getlen() { return longitud; }
};

// Crea una matriz_segura de tipo T con una longitud longi.
// La variable longi es un parámetro de plantilla sin tipo.
template <class T, int longi>  matriz_segura<T, longi>::matriz_segura() {
  // Inicializa los elementos de matriz a su valor predeterminado.
  for(int i=0; i < longi; ++i) mz[i] = T();
  longitud = longi;
}

// Devuelve una referencia al elemento del índice especificado.
// Proporciona revisión de rango para evitar errores de límite.
template <class T, int longi> T &matriz_segura<T, longi>::operator[](int i)
{
  if(i < 0 || i > longi-1) {
    // Toma aquí la acción apropiada. Esto es sólo
    // un marcador de posición de respuesta.
    cout << "\nEl valor " << i << " del \u00a1ndice queda fuera del l\u00a1mite.\n";
    exit(1);
  }
  return mz[i];
}

// Esto es una clase simple usada para demostrar una matriz de objetos.
// Observe que el constructor predeterminado da a x el valor -1.
class miclase {
public:
  int x;
  miclase(int i) { x = i; };
  miclase() { x = -1; }
};

int main()
```

```
{
  matriz_segura<int, 10> mz_int;   // matriz de entero de tamaño 10
  matriz_segura<double, 5> mz_double; // matriz double de tamaño 15
  int i;

  cout << "Valores iniciales de mz_int: ";
  for(i=0; i < mz_int.getlen(); ++i) cout << mz_int[i] << "  ";
  cout << endl;

  // Cambia los valores en mz_int.
  for(i=0; i < mz_int.getlen(); ++i) mz_int[i] = i;

  cout << "Nuevos valores para mz_int: ";
  for(i=0; i < mz_int.getlen(); ++i) cout << mz_int[i] << "  ";
  cout << "\n\n";

  cout << "Valores iniciales para mz_double: ";
  for(i=0; i < mz_double.getlen(); ++i) cout << mz_double[i] << "  ";
  cout << endl;

  // Cambia los valores en mz_double.
  for(i=0; i < mz_double.getlen(); ++i) mz_double[i] = (double) i/3;

  cout << "Nuevos valores para mz_double: ";
  for(i=0; i < mz_double.getlen(); ++i) cout << mz_double[i] << "  ";
  cout << "\n\n";;

  // matriz_segura también trabaja con objetos.
  matriz_segura<miclase, 3> mc_mz; // miclase array of size 3

  cout << "Valores iniciales en mc_mz: ";
  for(i = 0; i < mc_mz.getlen(); ++i) cout << mc_mz[i].x << " ";
  cout << endl;

  // Da algunos valores a mc_mz.
  mc_mz[0].x = 19;
  mc_mz[1].x = 99;
  mc_mz[2].x = -97;

  cout << "Nuevos valores para mc_mz: ";
  for(i = 0; i < mc_mz.getlen(); ++i) cout << mc_mz[i].x << " ";
  cout << endl;

  // Esto crea un desbordamiento de límite.
  mz_int[12] = 100;

  // Convierta en comentario la línea anterior y luego quite las marcas
  // de comentario de la línea siguiente para quedarse corto del límite.
//  mz_int[-2] = 100;

  return 0;
}
```

Aquí se muestra la salida:

```
Valores iniciales de mz_int: 0  0  0  0  0  0  0  0  0  0
Nuevos valores para mz_int: 0  1  2  3  4  5  6  7  8  9

Valores iniciales para mz_double: 0  0  0  0  0
Nuevos valores para mz_double: 0  0.333333  0.666667  1  1.33333

Valores iniciales en mc_mz: -1 -1 -1
Nuevos valores para mc_mz: 19 99 -97

El valor 12 del índice queda fuera del límite.
```

En el programa, preste especial atención a esta instrucción:

```
mz_int[12] = 100;
```

Trata de asignar 100 a la ubicación 12 dentro de **mz_int**. ¡Pero ésta sólo tiene 10 elementos de largo! Si fuera una matriz normal, entonces ocurriría un desbordamiento de límite. Por fortuna, en este caso, el intento es interceptado por **operator[]()** y el programa se termina antes de que pueda hacerse cualquier daño. (En la práctica real, se proporcionaría alguna especie de manejo de errores para tratar con la condición de fuera del rango; no sería necesario que el programa terminara.)

Opciones

Aunque la sobrecarga del operador de subíndice suele ser el mejor método en casos en que se aplica el concepto de "subíndice", en ocasiones verá que se usan, en cambio, funciones "get" y "put". En este caso, el índice del elemento deseado se pasa a la función "get" o "put" explícitamente como un argumento. Por ejemplo, podría usarse la siguiente secuencia para obtener la tercera cadena o para establecer la cuarta en algún conjunto de valores de cadena:

```
cad = get(3);
put(4, "probando");
```

Por supuesto, el subíndice ofrece un método más limpio, pero el método "get" y "put" es común en código C heredado. Si encuentra este tipo de código, tal vez quiera actualizarlo a C++ al sobrecargar a **[]**.

Sobrecargue el operador –>

Componentes clave		
Encabezados	**Clases**	**Funciones**
		tipo *operator–>()

Uno de los operadores más interesantes es –>. Se le denomina el operador de *acceso a miembro de clase*. Es un operador unario que devuelve un apuntador. Éste se relaciona de una manera u otra con el objeto en que se invoca a –>. La naturaleza precisa de la relación está definida por la clase para la que está definido –>. En cuanto a su relación con la sobrecarga, el –> es la fuente de muchas preguntas (y, en ocasiones, de confusión). En esta solución se demuestra cómo sobrecargarlo. Se incluye un ejemplo adicional que muestra la manera en que puede usarse un –> sobrecargado para crear un "apuntador seguro".

Paso a paso

La sobrecarga del operador –> incluye los pasos siguientes:

1. El operador de acceso a miembros debe ser un miembro no estático de la clase para la que está definido. No puede ser una función que no sea miembro. Por tanto, agregue **operator->()** como miembro a la clase en que estará operando.
2. Dentro de la función, obtenga un apuntador al objeto que invoca, o asociado de alguna manera con él.
3. Devuelva el apuntador.

Análisis

El operador –> está sobrecargado como operador unario. Aquí se muestra su uso general:

objeto–>elemento

Aquí, *objeto* es el objeto que activa la llamada. Ésta debe ser una instancia de la clase para la que está definido el operador de acceso a miembros, y se pasa a **operator–>()** mediante **this**. El *elemento* debe ser algún miembro accesible dentro del objeto. La función debe devolver un apuntador a *objeto* o a un objeto administrado por *objeto*. El uso principal del operador de acceso a miembro es dar soporte a lo que se considera "apuntadores seguros" o "apuntadores inteligentes". Se trata de apuntadores que verifican la integridad de un apuntador antes de realizar una acción con él. Otros usos incluyen la creación de apuntadores que administran automáticamente la memoria o que dan soporte a la recolección de basura.

La forma general de un **operator–>()** se muestra a continuación:

tipo *operator–>() {
  // Devuelve un apuntador al objeto que invoca.
}

Aquí, *tipo* debe ser el mismo que la clase para la que el **operator–>** está definido. Una función **operator–>()** debe ser un miembro no estático de su clase.

Ejemplo

En el siguiente ejemplo se muestra cómo sobrecargar **–>**. Simplemente devuelve un apuntador al objeto que invoca. Esto permite el uso de **–>** para acceder a un miembro de **miclase** a través de un objeto, en lugar de un apuntador a un objeto. Por tanto, la sobrecarga de **operator–>** hace que los operadores **–>** y **.** sean equivalentes. Aunque este ejemplo es útil para ilustrar el efecto de la sobrecarga de **–>**, porque es muy corto, no representa un buen uso (ni una práctica recomendada). Para ver la manera en que se emplearía normalmente un **–>** sobrecargado, consulte el Ejemplo adicional.

```
// Demuestra operator->().

#include <iostream>

using namespace std;

class miclase {
public:
  int i;

  // Sobrecarga -> para regresar un apuntador al objeto que invoca.
  miclase *operator->() { return this; }
};

int main()
{
  miclase ob;

  ob->i = 10; // igual que ob.i

  cout << ob.i << " " << ob->i;

  return 0;
}
```

Aquí se muestra la salida:

```
10 10
```

Ejemplo adicional: una clase simple de apuntador seguro

Aunque en el ejemplo anterior se presenta el mecanismo para sobrecargar **–>**, no se muestra toda su capacidad. Por lo general, el **–>** está sobrecargado para implementar un tipo de apuntador personalizado que restringe o monitorea, de una manera u otra, acciones sobre el apuntador. Por ejemplo, podría crear un tipo de apuntador que proporcione recolección automática de basura. Sin embargo, tal vez el uso más común sea crear un "apuntador seguro" que evite acciones no válidas mediante el apuntador, como dejar de referenciar o acceder a un miembro mediante un apuntador nulo. Este tipo de apuntador puede implementarse al sobrecargar los operadores * y **–>** para que confirmen que el apuntador no es nulo antes de proceder con la operación. Una implementación simple de este concepto se desarrolló en este ejemplo.

En el siguiente programa se crea una clase de apuntador seguro simple llamada **apt_seguro** que evita operaciones sobre un apuntador nulo. Hace esto al sobrecargar **–>** y *. (Cuando se usa como operador para dejar de hacer referencia, el * se sobrecarga como operador unario.) Estos operadores están sobrecargados para evitar que deje de hacerse referencia a un apuntador nulo o que se use para acceder a un miembro.

La clase **apt_seguro** se implementa como clase de plantilla en que el parámetro de tipo especifica el tipo base del apuntador. Por ejemplo, para crear un apuntador seguro a un **int**, use esta declaración:

```
apt_seguro<int> aptint;
```

Una vez que haya creado el apuntador seguro, puede usarlo como uno normal. Por ejemplo, puede asignarle una dirección de un objeto en memoria con la siguiente instrucción:

```
aptint = new int;
```

Puede establecer u obtener el valor del objeto mediante el apuntador al usar el operador *. Por ejemplo:

```
*aptint = 23;
```

En el caso de apuntadores a objetos de clase, puede usar **–>** para acceder a un miembro. En el caso de ambos operadores, **apt_seguro** confirma que el apuntador no sea nulo antes de aplicar el * o **–>**.

La clase **apt_seguro** funciona al encapsular un apuntador en un campo llamado **apt**. Se trata de un miembro privado, y el acceso a él sólo está permitido mediante operadores sobrecargados, incluido el operador de asignación sobrecargado. También se proporciona una función de conversión, que provee una conversión de **apt_seguro** a **T** *. Esto permite que se use un **apt_seguro** como operando para el operador **delete**, por ejemplo.

Si se hace un intento de usar un apuntador nulo, los operadores sobrecargados * y **–>** lanzarán un objeto de tipo **apt_malo**, que es una clase de excepción personalizada. El código que usa **apt_seguro** necesitará revisar esa excepción.

En el siguiente programa se incluyen las clases **apt_seguro** y **apt_malo**. También se define una clase llamada **miclase**, que se usa para demostrar **–>** con un **apt_seguro**. Aunque es muy simple, **apt_seguro** le da una idea de la capacidad de sobrecargar el operador **–>** y de crear sus propios tipos de apuntador. Los personalizados pueden ser muy útiles para evitar errores o para implementar esquemas personalizados de administración de memoria. Esté consciente de que, por supuesto, un tipo de apuntador personalizado siempre será más lento que uno simple, debido al trabajo adicional que introduce su código.

```
// Demuestra una clase muy simple de apuntador seguro.

#include <iostream>
#include <string>

using namespace std;

// El tipo de excepción lanzado por el apuntador seguro.
class apt_malo {
public:
  string msj;

  apt_malo(string cad) { msj = cad; }
};

// Una clase usada para demostrar el apuntador seguro.
class miclase {
public:
  int alfa;
  int beta;
  miclase(int p, int q) { alfa = p; beta = q; }
};

// Una clase muy simple de "apuntador seguro" que confirma
```

```
// que un apuntador señale a algún lado antes de usarse.
//
// El parámetro de plantilla T especifica el tipo de base
// del apuntador.
//
// Nota: esta clase sólo sirve para demostración. Sólo está
// orientada a ejemplificar la sobrecarga del operador ->.
// Una clase de apuntador seguro adecuado para trabajo real
// tiene que ser más completa y más resistente.
//
template <class T> class apt_seguro {
  T *apt;
public:
  apt_seguro() { apt = 0; }

  // Sobrecarga -> para que evite un intento de usar un apuntador
  // nulo para acceder a un miembro.
  T *operator->() {
    if(!apt != 0) throw apt_malo("Intento de usar -> en un apuntador nulo.");
    else return apt;
  }

  // Sobrecarga el operador de apuntador unario *. Este operador
  // evita que se elimine la referencia a un apuntador nulo.
  T &operator*() {
    if(!apt) throw apt_malo("Intento de dejar de hacer referencia a un apuntador
nulo.");
    else return *apt;
  }

  // Conversión de apt_seguro a T *.
  operator T *() { return apt; }

  T *operator=(T *val) { apt = val; return apt; }
};

int main()
{
  // Primero, usa apt_seguro en un entero.

  apt_seguro<int> aptint;

  // Genera una excepción al tratar de usar un apuntador
  // antes de que señale a algún objeto.
  try {
    *aptint = 23;
    cout << "El valor al que apunta aptint es: " << *aptint << endl;
  } catch(apt_malo bp) {
    cout << bp.msj << endl;
  }

  // Apunta aptint a un objeto.
  aptint = new int;
```

```
  // Ahora sí funcionará la siguiente secuencia.
  try {
    *aptint = 23;
    cout << "El valor al que apunta aptint es: " << *aptint << "\n\n";
  } catch(apt_malo bp) {
    cout << bp.msj << endl;
  }

  // Ahora, usa apt_seguro en una clase.

  apt_seguro<miclase> aptmc;

  // Esta secuencia trabajará de manera correcta.
  try {
    aptmc = new miclase(100, 200);
    cout << "Los valores de alfa y beta para aptmc son: "
         << aptmc->alfa << " y " << aptmc->beta << endl;

    aptmc->alfa = 27;
    cout << "Nuevo valor para aptmc->alfa: " << aptmc->alfa << endl;
    cout << "Igual que (*aptmc).alfa: " << (*aptmc).alfa << endl;

    aptmc->beta = 99;
    cout << "Nuevo valor para aptmc->beta: " << aptmc->beta << "\n\n";
  } catch(apt_malo bp) {
    cout << bp.msj << endl;
  }

  // Crea otro apuntador de miclase pointer, pero no lo inicializa.
  apt_seguro<miclase> aptmc2;

  // La siguiente asignación lanzará una excepción porque aptmc2
  // no señala a algún lado.
  try {
    aptmc2->alfa = 88;
  } catch(apt_malo bp) {
    cout << bp.msj << endl;
  }

  delete aptint;
  delete aptmc;

  return 0;
}
```

Aquí se muestra la salida:

```
Intento de dejar de hacer referencia a un apuntador nulo.
El valor al que apunta aptint es: 23

Los valores de alfa y beta para aptmc son: 100 y 200
Nuevo valor para aptmc->alfa: 27
Igual que (*aptmc).alfa: 27
Nuevo valor para aptmc->beta: 99

Intento de usar -> en un apuntador nulo.
```

Opciones

Tenga cuidado cuando sobrecargue **–>**. Los apuntadores ya son una característica problemática para algunos programadores. Si sobrecarga un **–>** de una manera confusa, poco intuitiva, simplemente alterará la estructura de su código y dificultará su mantenimiento. En general, sólo debe sobrecargar **–>** cuando cree un tipo de apuntador personalizado. Más aún, su tipo personalizado debe actuar y tener el aspecto de un apuntador normal. En otras palabras, su operación debe ser transparente y tener un uso consistente con el de un apuntador integrado. Por supuesto, su tipo de apuntador puede realizar revisiones adicionales o implementar un esquema de administración de memoria personalizado, pero debe funcionar como un apuntador normal cuando se usa en un programa.

En algunos casos, tal vez encuentre que C++ ya provee el tipo de apuntador que desea. Por ejemplo, una clase que a menudo se pasa por alto y que es proporcionada por la biblioteca estándar de C++ es **auto_ptr**, que libera automáticamente la memoria a la que señala cuando el apuntador sale del ámbito. Consulte *Use **auto_ptr*** para conocer más detalles.

Sobrecargue new y delete

Componentes clave		
Encabezados	**Clases**	**Funciones**
		void operator delete(void *apt) void operator delete[](void *apt) void *operator new(size_t tam) void *operator new[](size_t tam)

Los recién llegados a C++ se sorprenden, en ocasiones, al aprender que ahora **new** y **delete** se consideran operadores. Como tales, es posible sobrecargarlos. Tal vez decida hacerlo si quiere usar algún método de asignación especial. Por ejemplo, tal vez quiera rutinas de asignación que empiecen automáticamente a usar un archivo de disco como memoria virtual cuando el heap se haya agotado. O tal vez desee usar un esquema de asignación basado en la recolección de basura. Cada vez que lo necesite, es relativamente fácil sobrecargar estos operadores, y en esta solución se muestra el proceso.

Paso a paso

Para sobrecargar **new** y **delete** se necesitan estos pasos:

1. Para sobrecargar **new** para objetos individuales, implemente **operator new()**. Haga que devuelva un apuntador a un bloque de memoria que sea lo suficientemente grande como para contener el objeto.
2. Para sobrecargar **new** para matrices de objetos, implemente **operator new[]()**. Haga que devuelva un apuntador a un bloque de memoria que sea lo suficientemente grande como para contener la matriz.
3. Para sobrecargar **delete** para un objeto individual, implemente **operator delete()**. Haga que libere la memoria usada por el objeto.

4. Para sobrecargar **delete** para un apuntador a una matriz, implemente **operator delete[]()**. Haga que libere la memoria usada por el objeto.

Análisis

Antes de empezar, necesita dejarse en claro un tema importante. Los operadores **new** y **delete** pueden sobrecargarse globalmente o en relación con una clase específica. Cuando se sobrecargan de manera global, la nueva versión de **new** y **delete** reemplaza a las versiones predeterminadas cuando se asigna memoria a los tipos integrados y a cualquier clase que no proporcione su propia sobrecarga de **new** y **delete**. Por desgracia, en ocasiones esto causa efectos colaterales indeseables. Por ejemplo, código de tercero podría usar **new** y **delete** de una manera incompatible con las versiones sobrecargadas. Por esto, no se recomienda la sobrecarga global de **new** y **delete**, excepto en casos raros. En cambio, se recomienda la sobrecarga clase por clase. Cuando **new** y **delete** son sobrecargados por una clase, sólo se usan cuando se asigna memoria para objetos de la clase. Esto elimina la posibilidad de efectos colaterales fuera de la clase. Éste es el método usado en esta solución, y en el siguiente análisis se supone que se les está sobrecargando en relación con una clase mediante el uso de funciones miembro.

Hay dos formas básicas de **new** y **delete**. La primera es para asignación y liberación de objetos individuales. La segunda para las de matrices de objetos. Ambas formas pueden sobrecargarse y ambas se describen aquí. Empezaremos con las formas para objetos individuales.

He aquí las formas generales de **new** y **delete** sobrecargadas para objetos individuales:

```
// Asigna memoria a un objeto.
void *operator new(size_t tam)
{
  // Asigna memoria para el objeto y devuelve un apuntador a
  // la memoria. El tamaño en bytes del objeto se pasa en tam.
  // Lanza bad_alloc si falla.
}

// Libera memoria previamente asignada.
void operator delete(void *apt)
{
  // Libera la memoria a la que señala apt.
}
```

El parámetro *tam* contendrá el número de bytes necesarios para contener el objeto que se está asignando. Es la cantidad de memoria que su versión de **new** debe asignar (**size_t** es un **typedef** para alguna forma de entero sin signo). La función **new** sobrecargada debe devolver un apuntador a la memoria que se asigna, o lanzar una excepción **bad_alloc** si ocurre un error de asignación. Más allá de estas restricciones, el **new** sobrecargado puede hacer todo lo demás que necesite. Cuando asigne un objeto usando **new** (sea su propia versión o no), se llamará automáticamente al constructor del objeto.

La función **delete** recibe un apuntador a la región de la memoria que habrá de liberarse. Debe regresarse al sistema la memoria previamente asignada. Cuando se elimina un objeto, se llama automáticamente a su destructor. Es importante que **delete** sólo se use en un apuntador que se asignó previamente mediante **new**.

Si quiere tener la capacidad de asignar matrices a objetos empleando su propio sistema de asignación, necesitará sobrecargar **new[]** y **delete[]**, que son las formas de matriz de **new** y **delete**. He aquí las formas generales:

```
// Asigna una matriz de objetos.
void *operator new[](size_t tam)
{
  // Asigna memoria para la matriz y devuelve un apuntador a
  // ella. El número de bytes que se asignará se pasa en tam.
  // Lanza bad_alloc si falla.
}

// Elimina una matriz de objetos.
void operator delete[](void *apt)
{
  // Libera la memoria a la que señala apt.
}
```

Cuando se asigna una matriz, se llama automáticamente al constructor de cada objeto en ella. Cuando se libera una matriz, se llama automáticamente al destructor de cada objeto. No tiene que proporcionar código explícito para completar estas acciones.

Ejemplo

En el siguiente ejemplo se sobrecargan **news** y **delete** para la clase **tres_d**. Se sobrecargan las formas de objeto y de matriz de cada una. Para simplificar el ejemplo, no se usa un nuevo esquema de asignación. En cambio, los operadores sobrecargados simplemente invocarán las funciones de la biblioteca estándar de C **malloc()** y **free()**. La función **malloc()** asigna un número específico de bytes y devuelve un apuntador a ellos. Devuelve null si no es posible asignar la memoria. Dado un apuntador a memoria previamente asignado por **malloc()**, **free()** libera la memoria, dejándola disponible para usarla de nuevo. En general **malloc()** y **free()** tienen una funcionalidad similar a la de **new** y **delete**, pero de una manera más depurada.

```
// Sobrecarga new, new[], delete y delete[] para la clase tres_d.
//
// Este programa usa las funciones de C malloc() y free()
// para asignar y liberar memoria dinámica. Requieren el
// encabezado <cstdlib>.

#include <iostream>
#include <cstdlib>
#include <new>

using namespace std;

// Una clase que encapsula coordenadas tridimensionales.
class tres_d {
  int x, y, z; // Coordenadas 3-D
public:
  tres_d() { x = y = z = 0; }
  tres_d(int i, int j, int k) { x = i; y = j; z = k; }

  // Establece las coordenadas de un objeto después de crearlo.
```

```
  void set(int i, int j, int k) { x = i; y = j; z = k; }

  // Sobrecarga new y delete para objetos de tres_d.
  void *operator new(size_t tam);
  void operator delete(void *apt);

  // Sobrecarga new[] y delete[] para matrices de tres_d.
  void *operator new[](size_t tam);
  void operator delete[](void *apt);

  // Hace que el insertador sobrecargado sea un amigo.
  friend ostream &operator<<(ostream &flujo, tres_d op);
};

// El insertador de tres_d es una función de operador no miembro.
ostream &operator<<(ostream &flujo, tres_d op) {
  flujo << op.x << ", " << op.y << ", " << op.z << endl;

  return flujo;
}

// Sobrecarga new para tres_d.
void *tres_d::operator new(size_t tam)
{
  void *apt;

  cout << "Usando new sobrecargado para tres_d.\n";
  apt =  malloc(tam);
  if(!apt) {
    bad_alloc ba;
    throw ba;
  }
  return apt;
}

// Sobrecarga delete para tres_d.
void tres_d::operator delete(void *apt)
{
  cout << "Usando delete sobrecargado para tres_d.\n";
  free(apt);
}

// Sobrecarga new[] para matrices de tres_d.
void *tres_d::operator new[](size_t tam)
{
  void *apt;

  cout << "Usando new[] sobrecargado para tres_d.\n";
  apt =  malloc(tam);
  if(!apt) {
    bad_alloc ba;
    throw ba;
  }
  return apt;
```

```
}

// Sobrecarga delete[] para matrices de tres_d.
void tres_d::operator delete[](void *apt)
{
  cout << "Usando delete[] sobrecargado para tres_d.\n";
  free(apt);
}

int main()
{
  tres_d *a1, *a2;
  int i;

  // Asigna un objeto de tres_d.
  try {
    a1 = new tres_d (10, 20, 30);
  } catch (bad_alloc xa) {
    cout << "Error de asignación para a1.\n";
    return 1;
  }

  cout << "Coordenadas del nuevo objeto al que apunta a1: " << *a1;

  // Libera el objeto.
  delete a1;

  cout << endl;

  // Asigna una matriz de tres_d.
  try {
    a2 = new tres_d [10]; // asigna una matriz
  } catch (bad_alloc xa) {
    cout << "Error de asignación para a2.\n";
    return 1;
  }

  // Asigna coordenadas a tres de los elementos de a2.
  a2[1].set(99, 88, 77);
  a2[5].set(-1, -2, -3);
  a2[8].set(56, 47, 19);

  cout << "Contenido de una matriz din\u00a0mica de tres_d:\n";
  for(i=0; i<10; i++) cout << a2[i];

  // Libera la matriz.
  delete [] a2;

  return 0;
}
```

Aquí se muestra la salida:

```
Usando new sobrecargado para tres_d.
Coordenadas del nuevo objeto al que apunta a1: 10, 20, 30
Usando delete sobrecargado para tres_d.

Usando new[] sobrecargado para tres_d.
Contenido de una matriz dinámica de tres_d:
0, 0, 0
99, 88, 77
0, 0, 0
0, 0, 0
0, 0, 0
-1, -2, -3
0, 0, 0
0, 0, 0
56, 47, 19
0, 0, 0
Usando delete[] sobrecargado para tres_d.
```

Opciones

C++ da soporte a una versión "sin lanzamiento de excepciones" de **new**. Esta opción hace que **new** actúe como lo hacía en versiones anteriores de C++, en que devolvía null si no era posible asignarle memoria. (Las versiones modernas de C++ lanzan una excepción **bad_alloc** cuando **new** falla.) Puede crear versiones sobrecargadas de las versiones que no lanzan excepciones al usar estas formas de **operator new()** y **operator new[]()**:

```
// Versión sin lanzamiento de excepciones de new.
void *operator new(size_t tam, const nothrow_t &no usada)
{
  // Asigna la memoria al objeto. Si tiene éxito, devuelve
  // un apuntador a la memoria. De lo contrario, devuelve null.
}

// Versión sin lanzamiento de excepciones de new[].
void *operator new[](size_t tam, const nothrow_t &no usada)
{
  // Asigna la memoria para la matriz. Si tiene éxito, devuelve
  // un apuntador a la memoria. De lo contrario, devuelve null.
}
```

El tipo **nothrow_t** está definido en **<new>**.

Cuando use la versión sin lanzamiento de excepciones, especifique el objeto **nothrow** en la llamada a **new** y revise un valor de devolución nulo, como se muestra aquí:

```
apt = new(nothrow) int;
if(!apt) {
  cout << "Ha fallado la asignaci\u00a2n.\n";
  // maneja la falla ...
}
```

El objeto **nothrow** es una instancia de **nothrow_t** y es proporcionado por **<new>**.

Sobrecargue los operadores de aumento y disminución

Componentes clave		
Encabezados	**Clases**	**Funciones**
		tipo-ret operator++() *tipo-ret* operator++(int *no_usada*) *tipo-ret* operator–() *tipo-ret* operator–(int *no_usada*)

En cuanto a la sobrecarga de operadores, **++** (aumento) y **– –** (reducción) generan la mayor parte de las preguntas. Aunque ninguno de los dos es difícil de sobrecargar, resulta fácil hacerlo de manera ligeramente equivocada, lo que lleva a que el operador trabaje de manera correcta en algunos casos, pero que falle en otros. Esto puede dar como resultado errores difíciles de diagnosticar. Los operadores de aumento y reducción también tienen dos formas diferentes, de prefijo y sufijo, y ambas deben sobrecargarse para que el operador siempre funcione correctamente. En esta solución se muestra cómo manejar estos operadores que, en ocasiones, resultan problemáticos.

Paso a paso

Para sobrecargar los operadores de aumento y reducción empleando funciones miembro, se requieren estos pasos:

1. Para sobrecargar la forma de prefijo del operador de aumento, cree una función **operator++()**. Dentro de esa función, aumente el objeto que invoca y devuelva el resultado.
2. Para sobrecargar la forma de sufijo del operador de aumento, cree una función **operator++(int)**. Dentro de esa función, cree un objeto temporal que contenga el valor original del operando. Luego, aumente el objeto que invoca. Por último, devuelva el valor original.
3. Para sobrecargar la forma de prefijo del operador de reducción, cree una función **operator– –()**. Dentro de esa función, reduzca el objeto que invoca y devuelva el resultado.
4. Para sobrecargar la forma de sufijo del operador de reducción, cree una función **operator – –(int)**. Dentro de esa función, cree un objeto temporal que contenga el valor original del operando. Luego, reduzca el objeto que invoca. Por último, devuelva el valor original.

Análisis

Hay dos formas de los operadores **++** y **– –**: prefijo y sufijo. La forma de prefijo aumenta el operando y devuelve el resultado. La forma de sufijo almacena el valor inicial del operando, aumenta éste y luego regresa el valor original. Ambas formas pueden sobrecargarse, y cada una es sobrecargada por su propia función.

Con mayor frecuencia, los operadores de aumento y reducción son funciones miembros de la clase para la que están definidos. Éste es el método usado en esta solución. Sin embargo, también pueden implementarse mediante funciones que no son miembro, y esto se describe en la sección *Opciones* de esta solución.

He aquí las formas generales de **operator++()** y **operator– –()** cuando se implementan como funciones miembro. Se muestran las formas de prefijo y sufijo:

```
// Aumento de prefijo
tipo-ret operator++() {
  // Aumenta el operando y devuelve el resultado.
}

// Aumento de sufijo
tipo-ret operator++(int no_usada) {
  // Almacena una copia del valor original de un operando.
  // Luego aumenta el operando.
  // Por último, devuelve el valor original.
}

// Reducción de prefijo
tipo-ret operator--() {
  // Reduce el operando y devuelve el resultado.
}

// Aumento de sufijo
tipo-ret operator--(int no_usada) {
// Almacena una copia del valor original de un operando.
  // Luego reduce el operando.
  // Por último, devuelve el valor original.
}
```

Preste especial atención al parámetro *no usada*. Suele ser cero y, por lo general, no se usa dentro de la función. Es simplemente una manera en que C++ indica a cuál función llamar.

Hay tres claves para sobrecargar correctamente el aumento y la reducción:

- Debe sobrecargar las formas de prefijo y sufijo.
- Cuando implemente la forma de prefijo, primero debe aumentar o reducir el valor y luego devolver el valor modificado.
- Cuando implemente la forma de sufijo, recuerde que debe almacenar el valor inicial y luego devolver ese valor. No devuelva por accidente el valor modificado.

Si sigue estas reglas, sus operadores de aumento y reducción se comportarán como los integrados. Si no las sigue, puede incurrir en problemas. Por ejemplo, si no sobrecarga las formas de prefijo y sufijo de un operador, entonces no podrá usarse la forma que no sobrecargue. Más aún, si no sobrecarga la forma de sufijo, algunos compiladores reportarán un error si trata de usar el operador de sufijo, y no compilarán su programa. Sin embargo, otros compiladores simplemente lanzarán una advertencia y luego usarán, en cambio, la forma de prefijo. Esto hará que el operador de sufijo actúe de manera inesperada.

Ejemplo

En el siguiente ejemplo se sobrecargan los operadores de aumento y reducción para la clase **tres_d**. Se proporcionan las formas de prefijo y sufijo.

```
// Sobrecarga ++ y -- para tres_d.

#include <iostream>

using namespace std;

// Una clase que encapsula coordenadas tridimensionales.
class tres_d {
  int x, y, z; // 3-D coordinates
public:
  tres_d() { x = y = z = 0; }
  tres_d(int i, int j, int k) { x = i; y = j; z = k; }

  // Sobrecarga ++ y  --. Proporciona formas de prefijo y sufijo.
  tres_d operator++(); // prefijo
  tres_d operator++(int nousada); // sufijo

  tres_d operator--(); // prefijo
  tres_d operator--(int nousada); // sufijo

  // Hace que el insertador sobrecargado sea un amigo.
  friend ostream &operator<<(ostream &flujo, tres_d op);
};

// Sobrecarga ++ de prefijo para tres_d.
tres_d tres_d::operator++() {
  x++;
  y++;
  z++;

  return *this;
}

// Sobrecarga ++ de sufijo para tres_d.
tres_d tres_d::operator++(int nousada) {
  tres_d temp = *this;

  x++;
  y++;
  z++;

  return temp;
}

// Sobrecarga -- de prefijo para tres_d.
tres_d tres_d::operator--() {
  x--;
  y--;
  z--;
```

```
  return *this;
}

// Sobrecarga -- de sufijo para tres_d.
tres_d tres_d::operator--(int nousada) {
  tres_d temp = *this;

  x--;
  y--;
  z--;

  return temp;
}

// El insertador de tres_d es una función de operador que no es miembro.
ostream &operator<<(ostream &flujo, tres_d op) {
  flujo << op.x << ", " << op.y << ", " << op.z << endl;

  return flujo;
}

int main()
{
  tres_d objA(1, 2, 3), objB(10, 10, 10), objC;

  cout << "Valor original de objA: " << objA;
  cout << "Valor original de objB: " << objB;

  // Demuestra ++ y -- como operaciones independientes.
  ++objA;
  ++objB;

  cout << "++objA: " << objA;
  cout << "++objB: " << objB;

  --objA;
  --objB;

  cout << "--objA: " << objA;
  cout << "--objB: " << objB;

  objA++;
  objB++;

  cout << endl;

  cout << "objA++: " << objA;
  cout << "objB++: " << objB;

  objA--;
  objB--;

  cout << "objA--: " << objA;
  cout << "objB--: " << objB;
```

```
  cout << endl;

  // Ahora, demuestra la diferencia entre las formas
  // de prefijo y sufijo de ++ y --.

  objC = objA++;
  cout << "Luego de que objC = objA++\n  objC: " << objC <<"  objA: "
       << objA << endl;

  objC = objB--;
  cout << "Luego de que objC = objB--\n  objC: " << objC <<"  objB: "
       << objB << endl;

  objC = ++objA;
  cout << "Luego de que objC = ++objA\n  objC: " << objC <<"  objA: "
       << objA << endl;

  objC = --objB;
  cout << "Luego de que objC = --objB\n  objC: " << objC <<"  objB: "
       << objB << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Valor original de objA: 1, 2, 3
Valor original de objB: 10, 10, 10
++objA: 2, 3, 4
++objB: 11, 11, 11
--objA: 1, 2, 3
--objB: 10, 10, 10

objA++: 2, 3, 4
objB++: 11, 11, 11
objA--: 1, 2, 3
objB--: 10, 10, 10

Luego de que objC = objA++
  objC: 1, 2, 3
  objA: 2, 3, 4

Luego de que objC = objB--
  objC: 10, 10, 10
  objB: 9, 9, 9

Luego de que objC = ++objA
  objC: 3, 4, 5
  objA: 3, 4, 5

Luego de que objC = --objB
  objC: 8, 8, 8
  objB: 8, 8, 8
```

Opciones

Aunque el uso de funciones miembro para sobrecargar los operadores de aumento y reducción es el método más común, también puede usar funciones que no son miembro. Tal vez quiera hacer esto cuando sobrecarga el operador en relación con una enumeración, o cuando está definiendo el aumento y la reducción de acuerdo con una clase de la que no tiene el código fuente. Cualquiera que sea la razón, es una tarea fácil. A continuación se muestran las formas que no son miembro de los operadores de aumento y reducción:

```
// Aumento de prefijo
tipo-ret operator++(type &op) {
  // Aumenta el operando y devuelve el resultado.
}

// Aumento de sufijo
tipo-ret operator++(type &op, int no_usada) {
  // Almacena una copia del valor original del operando.
  // Luego aumenta el operando.
  // Por último, devuelve el valor original.
}

// Reducción de prefijo
tipo-ret operator– –(type &op) {
  // Reduce el operando y devuelve el resultado.
}

// Aumento de sufijo
tipo-ret operator– –(type &op, int no_usada) {
// Almacena una copia del valor original del operando.
  // Luego reduce el operando.
  // Por último, devuelve el valor original.
}
```

Observe que el operando se pasa mediante referencia. Esto es necesario para permitir que las funciones modifiquen el operando.

En general, cuando quiera aumentar o reducir un objeto, la sobrecarga de los operadores **++** y **– –** es el mejor método. Sin embargo, en algunos casos, tal vez el uso de las funciones sea mejor. Por ejemplo, puede crear una función llamada **inc()** que aumente un objeto y **dis()** que lo reduzca. Tal vez quiera hacer esto cuando no desea modificar el valor del objeto. La función **inc()** o **dis()** quizá no devuelva el nuevo valor, sino que deje el objeto sin modificación. También podría hacer esto al sobrecargar los operadores de aumento y reducción de manera que no modifiquen el operando, pero esto podría hacer que funcionen de una manera inconsistente con su semántica normal.

Debe tener cuidado cuando trabaje con programas heredados de C++ en que se utilizan los operadores de aumento y reducción. En versiones anteriores de C++, no era posible especificar versiones separadas de un **++** o – – sobrecargado. La forma de prefijo se usaba en ambas. Los compiladores modernos por lo general lanzarán una advertencia en esta situación, pero es mejor no contar con ella. Lo más adecuado es confirmar que el aumento y la reducción están sobrecargados de manera apropiada. Si no lo están, necesita actualizarlos.

Cree una función de conversión

Componentes clave		
Encabezados	**Clases**	**Funciones**
		operator *tipo-destino*()

En ocasiones, querrá usar un objeto de clase en una expresión que incluya otro tipo de datos. Aunque los operadores sobrecargados pueden proporcionar un medio de hacerlo, en ocasiones todo lo que realmente quiere es una simple conversión del tipo de la clase al de destino. Para manejar estos casos, C++ le permite crear una *función de conversión*. Ésta convierte automáticamente el tipo de una clase en el de destino. Esto hace que la de conversión sea una de las funciones más útiles de C++. Por desgracia, también es una de las características más subestimadas. En esta solución se muestra cómo crear una función de conversión. En el proceso, se arroja un poco de luz sobre sus capacidades a veces ignoradas.

Paso a paso

Para crear una función de conversión, se requieren estos pasos:

1. Para proporcionar una conversión del tipo de una clase a uno de destino, agregue una función de conversión a la clase. Una función de conversión se basa en la palabra clave **operator**, como se describe en el análisis siguiente.
2. Dentro de la función de conversión, convierta el objeto en el tipo de destino.
3. Devuelva el resultado, que debe ser un valor compatible con el tipo de destino.

Análisis

Una función de conversión usa la palabra clave **operator**. La forma general de una conversión se muestra a continuación:

```
operator tipo-destino() {
  //Crea un valor que contiene la conversión.
  return valor;
}
```

Aquí, *tipo-destino* es el tipo de destino al que está convirtiendo su clase, y *valor* es el resultado de la conversión. El objeto que se está convirtiendo se pasa mediante **this**. Las funciones de conversión devuelven datos de tipo *tipo-destino*, y no se permite ningún otro especificador de tipo de devolución. Además, no se pueden incluir parámetros. Una función de conversión debe ser miembro de la clase para la que se define. Las funciones de conversión se heredan y pueden ser virtuales.

Una vez que ha creado una función de conversión, puede usarse un objeto de su clase en expresiones del tipo de destino. Esto significa que puede operarse mediante operadores (sin tener que sobrecargarlos), siempre y cuando el tipo de la expresión sea igual que el del destino de la función de conversión. Más aún, una función de conversión le permite pasar un objeto como archivo a una función, siempre y cuando el tipo de parámetro sea igual que el del destino. Éstas son características poderosas, que pueden obtenerse casi sin esfuerzo de programación alguno.

La mejor manera de apreciar el poder de una función de conversión es trabajar con un ejemplo. Suponga la clase **tres_d** mostrada aquí:

```
class tres_d {
  int x, y, z; // Coordenadas 3-D
public:
  tres_d() { x = y = z = 0; }
  tres_d(int i, int j, int k) { x = i; y = j; z = k; }

  // ...
};
```

Puede crear una conversión para **int** al agregar la siguiente función como un miembro:

```
operator int() {  return x + y + z; }
```

Esto convierte un objeto de **tres_d** en un entero que contiene la suma de las coordenadas.

Suponiendo la conversión anterior, ahora la siguiente secuencia es válida:

```
tres_d objA(1, 2, 3), objB(-1, -2, -3);
int resultado;
resultado = 10 + objA;
```

Después de que esto se ejecuta, el resultado contendrá el valor 16(10+1+2+3). Debido a que 10 es un valor **int**, cuando se le agrega **objA**, se invoca automáticamente **operator int()** en **objA** para proporcionar la conversión.

Ejemplo

En el siguiente ejemplo se pone en acción el análisis anterior. Crea una conversión de **tres_d** a **int**. Luego usa esa conversión para emplear un objeto de **tres_d** en una expresión de entero y pasa objetos de **tres_d** como argumentos a funciones que especifican un parámetro de entero.

```
// Crea funciones de conversión para tres_d.

#include <iostream>

using namespace std;

// Una clase que encapsula coordenadas tridimensionales.
class tres_d {
  int x, y, z; // Coordenadas 3-D
public:
  tres_d() { x = y = z = 0; }
  tres_d(int i, int j, int k) { x = i; y = j; z = k; }

  // Una conversión a int.
  operator int() { return x + y + z; }

  // Hace que el insertador sobrecargado sea un amigo.
  friend ostream &operator<<(ostream &flujo, tres_d op);
};

// El insertador de tres_d es una función de operador no miembro.
```

```
ostream &operator<<(ostream &flujo, tres_d op) {
  flujo << op.x << ", " << op.y << ", " << op.z << endl;

  return flujo;
}

// Devuelve la negación de v.
int neg(int v) {
  return -v;
}

// Devuelve true si x es menor que y.
bool lt(int x, int y) {
  if(x < y) return true;
  return false;
}

int main()
{
  tres_d objA(1, 2, 3), objB(-1, -2, -3);
  int resultado;

  cout << "El valor de objA: " << objA;
  cout << "El valor de objB: " << objB;
  cout << endl;

  // Usa objA en una expresión int.
  cout << "Usa un objeto de tres_d en una expresi\u00a2n int: ";
  resultado = 10 + objA;
  cout << "10 + objA: " << resultado << "\n\n";

  // Pasa objA a una función que toma un argumento int.
  cout << "Pasa un objeto de tres_d a un par\u00a0metro int: ";
  resultado = neg(objA);
  cout << "neg(objA): " << resultado << "\n\n";

  cout << "Compara la suma de las coordenadas con el uso de lt(): ";
  if(lt(objA, objB))
    cout << "objA es menor que objB\n";
  else if(lt(objB, objA))
    cout << "objB es menor que objA\n";
  else
    cout << "objA y objB suman el mismo valor.\n";


  return 0;
}
```

Aquí se muestra la salida:

```
El valor de objA: 1, 2, 3
El valor de objB: -1, -2, -3

Usa un objeto de tres_d en una expresión int: 10 + objA: 16

Pasa un objeto de tres_d a un parámetro int: neg(objA): -6

Compara la suma de las coordenadas con el uso de lt(): objB es menor que objA
```

Opciones

Puede crear diferentes funciones de conversión para cubrir distintas necesidades. Por ejemplo, podría definir conversiones de **tres_d** a **int**, **double** o **long**. Cada una se aplicará de manera automática, determinada por el tipo de conversión necesario.

En algunos casos, en lugar de usar una función de conversión, puede obtener el mismo resultado (pero no tan fácilmente) al sobrecargar los operadores que estará usando. En el ejemplo anterior, podría sobrecargar el **+** para operaciones que requieren objetos de **tres_d** y enteros. Por supuesto, esto no permitiría aún que se pase un objeto de **tres_d** a una función que usa un parámetro **int**.

Cree un constructor de copia

Componentes clave		
Encabezados	**Clases**	**Funciones**
		nombreclase (const *nombreclase &obj*)

Una característica a menudo subestimada pero increíblemente importante de C++ es el *constructor de copias*. Éste define la manera en que se hace la copia de un objeto. Debido a que C++ proporciona automáticamente un constructor de copias predeterminado para una clase, no todas las clases necesitan definir uno de manera explícita. Sin embargo, en el caso de muchas clases, el constructor de copias predeterminado es insuficiente, y su uso causa problemas. Esto se debe a que el constructor predeterminado crea una copia idéntica del original. Si un objeto contiene un recurso, como un apuntador a memoria o un objeto de flujo de archivo, entonces si se hace una copia, ésta también podría señalar a la misma memoria o tratar de usar el mismo archivo. En casos como éste, pronto habrá problemas. La solución consiste en definir un constructor de copias explícito que duplique un objeto, pero evite el posible problema. Con este fin, en esta solución se describe cómo crear un constructor de copias y se revisan las circunstancias bajo las que se necesita.

Paso a paso

Para crear un constructor de copias, se necesitan estos pasos:

1. Cree un constructor para la clase que tome sólo un parámetro, que es una referencia al objeto que habrá de copiarse.
2. Dentro del constructor, copie el objeto de una manera compatible con la clase.

Análisis

Empecemos por examinar el problema que se pretende que resuelva el constructor de copias. Como opción predeterminada, cuando se usa un objeto para inicializar otro, se hace una copia del original campo por campo. En el caso de campos escalares (entre los que se incluyen los apuntadores), se tiene una copia idéntica, byte por byte, del campo. Aunque esto es perfectamente adecuado para muchos casos (y a menudo es exactamente lo que desea que suceda), hay situaciones en que no debe usarse una copia idéntica. Una de las más comunes es cuando un objeto usa memoria asignada dinámicamente. Por ejemplo, suponga una clase llamada **miclase** que utiliza este tipo de memoria para algún propósito y que, en un campo, se mantiene un apuntador a esta memoria. Más aún, suponga que esta memoria se asigna cuando se construye un objeto y se libera cuando se ejecuta su destructor. Por último, suponga un objeto de **miclase** llamado **A**, que se usa para inicializar **B**, como se muestra aquí:

```
miclase B = A;
```

Si se hace una copia idéntica de **A** y se asigna a **B**, entonces en lugar de que **B** contenga un apuntador a su propia porción de memoria asignada dinámicamente, estará usando la misma porción de memoria que **A**. Esto casi siempre llevará a problemas. Por ejemplo, cuando se destruyan **A** y **B**, ¡la misma porción de memoria se liberará dos veces! Una vez para **A** y una más para **B**.

Un tipo similar de problemas puede ocurrir de dos maneras adicionales. La primera ocurre cuando se hace una copia de un objeto y se pasa como argumento a una función. Este objeto sale del ámbito (y se destruye) cuando se devuelve la función. La segunda ocurre cuando se crea un objeto temporal como valor devuelto de una función. Como tal vez lo sepa, los objetos temporales se crean automáticamente para contener el valor devuelto por una función. Este objeto temporal sale automáticamente de ámbito después de que termina la expresión que contiene la llamada a la función. En ambos casos, si el objeto temporal actúa sobre un recurso, como a través de un apuntador o un archivo abierto, entonces esas acciones tendrán efectos secundarios. En el caso de **miclase**, esto daría como resultado que el mismo bloque de memoria se libere dos o más veces. Es evidente que esta situación debe evitarse.

Para resolver el tipo de problemas que se acaba de describir, C++ le permite crear un *constructor de copias* explícito para una clase. Se llama a éste cuando un objeto inicializa a otro. Todas las clases tienen un constructor de copias predeterminado, que produce una copia miembro por miembro. Cuando define su propio constructor de copias, éste se usa en lugar del predeterminado.

Antes de seguir adelante, es importante comprender que C++ define dos tipos distintos de situación en que el valor de un objeto se asigna a otro. El primero es la asignación. El segundo es la inicialización, que puede ocurrir de tres maneras:

- Cuando un objeto inicializa explícitamente otro, como en una declaración.
- Cuando se hace una copia de un objeto para pasarlo a una función.
- Cuando se genera un objeto temporal (con mayor frecuencia, como un valor devuelto).

El constructor de copias sólo se aplica a las inicializaciones. No lo hace a asignaciones.

La forma más general de un constructor de copias se muestra a continuación:

nombreclase (const *nombreclase* &*ob*) {

  // Cuerpo del constructor de copias.

}

Aquí, *obj* es una referencia al objeto que se encuentra en el lado derecho de la inicialización. Es permisible que un constructor de copias tenga parámetros adicionales, siempre y cuando cuente con argumentos predeterminados definidos para ellos. Sin embargo, en todos los casos, el primer parámetro debe ser una referencia al objeto que hace la inicialización. Esta referencia puede ser **const**, **volatile**, o ambas.

Una vez más, suponga una clase llamada **miclase** y un objeto de tipo **miclase** llamado **A**. Además, suponiendo que **func1()** toma un parámetro de **miclase** y que **func2()** devuelve un objeto de **miclase**, cada una de las siguientes instrucciones requiere inicialización:

```
miclase B = A; // A inicializando B
miclase B(A);  // A inicializando B
func1(A);      // A pasada como parámetro
A = func2();   // A recibiendo un objeto temporal, devuelto
```

En los tres primeros casos, una referencia a **A** se pasa al constructor de copias. En el cuarto, se le pasa una referencia al objeto devuelto por **func2()**.

Dentro de un constructor de copias, debe manejar manualmente la duplicación de cada campo dentro del objeto. Esto, por supuesto, le da oportunidad de evitar situaciones posiblemente dañinas. Por ejemplo, en **miclase** que se acaba de describir, el nuevo objeto **miclase** podría asignar su propia memoria. Esto permitiría que el original y la copia fueran objetos equivalentes, pero completamente separados. También evita el problema de que ambos objetos usen la misma memoria porque si un objeto libera la memoria, no se afectará al otro. Si es necesario, puede inicializarse la memoria para que incluya el mismo contenido que el original.

En algunos casos, los mismos problemas que pueden ocurrir cuando se hace una copia de un objeto también ocurren cuando un objeto se asigna a otro. La razón es que el operador de asignación predeterminado hace una copia idéntica miembro por miembro. Puede evitar problemas al sobrecargar **operator=()** para que maneje usted mismo el proceso de asignación. Consulte *Técnicas básicas de sobrecarga de operadores* para conocer detalles acerca de la sobrecarga de asignación.

Ejemplo

En el siguiente ejemplo se demuestra el constructor de copias. Aunque es muy simple, enseña de manera clara cuándo se llama o no a un proceso de copia. (Un uso práctico del constructor de copias se muestra en el Ejemplo adicional que sigue.)

```
// Demuestra un constructor de copias.

#include <iostream>
using namespace std;

// Esta clase declara un constructor de copias.
class muestra {
```

```
public:
  int v;

  // Constructor predeterminado.
  muestra() {
    v = 0;
    cout << "Dentro del constructor predeterminado.\n";
  }

  // Constructor con parámetros.
  muestra(int i) {
    v = i;
    cout << "Dentro del constructor con par\u00a0metros.\n";
  }

  // Constructor de copias.
  muestra(const muestra &obj) {
    v = obj.v;
    cout << "Dentro del constructor de copias.\n";
  }
};

// Pasa un objeto a una función. Se llama al constructor de
// copias cuando se crea un objeto temporal para contener el
// valor pasado a x.
int dosveces(muestra x) {
  return x.v * x.v;
}

// Devuelve un objeto de una función. Se llama al constructor
// de copias cuando se crea un temporal para el valor devuelto.
muestra original(int i) {
  muestra s(i);
  return s;
}


int main()
{
  cout << "Crea muest(8).\n";
  muestra muest(8);
  cout << "muest tiene el valor " << muest.v << endl;

  cout << endl;

  cout << "Crea muest2 y lo inicializa con muest.\n";
  muestra muest2 = muest;
  cout << "muest2 tiene el valor " << muest2.v << endl;

  cout << endl;

  cout << "Pasa muest a dosveces().\n";
  cout << "Resultado de dosveces(muest): " << dosveces(muest) << endl;
  cout << endl;
```

```
  cout << "Creando muest3.\n";
  muestra muest3;

  cout << endl;

  cout << "Ahora, asigna a muest3 el valor devuelto por original(10).\n";
  muest3 = original(10);
  cout << "muest3 ahora tiene el valor " << muest3.v << endl;

  cout << endl;

  // La asignación no invoca al constructor de copias.
  cout << "Ejecuta muest3 = muest.\n";
  muest3 = muest;
  cout << "Observe que no se ha usado el constructor de copias "
       << "para asignaciones.\n";

  return 0;
}
```

Aquí se muestra la salida:

```
Crea muest(8).
Dentro del constructor con parámetros.
muest tiene el valor 8

Crea muest2 y lo inicializa con muest.
Dentro del constructor de copias.
muest2 tiene el valor 8

Pasa muest a dosveces().
Dentro del constructor de copias.
Resultado de dosveces(muest): 64

Creando muest3.
Dentro del constructor predeterminado.

Ahora, asigna a muest3 el valor devuelto por original(10).
Dentro del constructor con parámetros.
Dentro del constructor de copias.
muest3 ahora tiene el valor 10

Ejecuta muest3 = muest.
Observe que no se ha usado el constructor de copias para asignaciones.
```

Como se observa, se llama al constructor de copias cuando un objeto inicializa a otro. No se le llama durante la asignación. Un tema adicional: la instrucción

```
muestra muest2 = muest;
```

también puede escribirse como

```
muestra muest2(muest);
```

Ambas formas dan como resultado que se use el constructor de copias para crear una copia de **muest**.

Ejemplo adicional: una matriz segura que usa asignación dinámica

En el ejemplo anterior se mostró claramente cuándo se llama o no a un constructor de copias. Sin embargo, no ilustra el tipo de situación en que uno es necesario. En este ejemplo sí se hace. Demuestra la necesidad del constructor de copias al desarrollar otra implementación de una "matriz segura", que es una que evita desbordamiento de límites o que se quede corto de éstos. El método usado aquí depende de que la memoria asignada dinámicamente contenga la matriz. Como verá, esta técnica requiere un constructor de copias explícito para evitar problemas.

Antes de empezar, resulta útil contrastar este método con el mostrado en *Sobrecargue el operador de subíndice* **[]**, en páginas anteriores de este capítulo. En esa solución, se creó en el ejemplo un tipo de matriz llamada **matriz_segura** que encapsulaba una matriz estática que contenía, en realidad, los elementos. Por tanto, cada **matriz_segura** era respaldada por una matriz estática de longitud completa. Como resultado, si se necesitaba una matriz segura muy larga, el objeto de **matriz_segura** resultante también era muy largo, porque encapsularía toda la matriz.

La versión desarrollada aquí utiliza un método diferente. La llamada **matriz_segura_din** asigna memoria dinámicamente a la matriz y almacena sólo un apuntador a esa memoria. Esto tiene la ventaja de hacer más pequeños los objetos de la matriz segura (mucho más pequeños, en algunos casos). Esto los hace más eficientes cuando se pasan a funciones, por ejemplo. Por supuesto, se requiere un poco más de trabajo para implementar una matriz segura que usa memoria dinámica, porque se necesitan un constructor de copias y un operador de asignación sobrecargado. Como **matriz_segura**, que se mostró antes, **matriz_segura_din** sobrecarga el operador de subíndice **[]** para permitir que los subíndices normales, tipo matriz, accedan a los elementos de la matriz.

La clase **matriz_segura_din** es genérica, lo que significa que puede usarse para crear cualquier tipo de matriz. El número de elementos en la matriz se pasa a un argumento sin tipo en su especificación de plantilla. Luego, su constructor asigna memoria suficiente para que contenga la matriz del tamaño y el tipo deseados. Un apuntador a esta memoria se almacena en **aptm**. El destructor para **matriz_segura_din** libera automáticamente esta memoria cuando un objeto sale del ámbito. De otra manera, como el **[]** está sobrecargado, puede usarse una **matriz_segura_din** como una matriz normal.

Cuando una **matriz_segura_din** se usa para inicializar otra, se llama al constructor de copias. Crea una copia del original al asignar primero memoria para la matriz y luego copiar elementos de la matriz original en la memoria recién asignada. De esta manera, cada **aptm** del objeto señala a su propia matriz. Sin el constructor de copias, se haría una copia idéntica de **matriz_segura_din**, lo que daría como resultado dos objetos con **aptm** que señalan a la misma memoria. Entre otros posibles problemas, esto daría como resultado un intento por liberar la misma memoria más de una vez cuando el objeto sale del ámbito. El constructor de copias evita esto.

El mismo tipo de problema que evita el constructor de copias también puede ocurrir cuando un objeto de **matriz_segura_din** se asigna a otro. Para evitar este problema, el operador de asignación también se sobrecarga para que el contenido de la matriz se copie, pero la memoria asignada dinámicamente usada por cada objeto permanezca separada.

Un último tema: el constructor de copias y el operador de asignación sobrecargada despliegan un mensaje cada vez que se les llama. Esto es simplemente para ejemplificación. Por lo general, ninguno generaría alguna salida.

```
// Una clase de matriz segura que evita errores de límite de matriz.
// Utiliza el operador de subíndice para acceder a los elementos de la
// matriz. Esta versión difiere del método utilizado en la solución:
//
//     Sobrecargue el operador de subíndice []
//
// porque asigna memoria a la matriz de manera dinámica en lugar de
// estática.
//
// Un constructor de copias explícito se implementa de modo que una
// copia de un objeto de matriz_segura usa su propia memoria asignada.
// Por tanto, el objeto original y la copia NO señalan a la misma
// memoria. El operador de asignación también se sobrecarga por la misma
// razón. En ambos casos, el contenido de la matriz se copia para que
// la matriz y la copia contengan los mismos valores.

#include <iostream>
#include <new>
#include <cstdlib>

using namespace std;

// Una clase de matriz segura que asigna memoria dinámicamente para la
// matriz. La longitud de la matriz se pasa como un argumento sin tipo
// en la especificación de plantilla.
template <class T, int longi> class matriz_segura_din {
  T *aptm;  // apuntador a la memoria que contiene la matriz
  int longitud; // número de elementos en la matriz
public:

  // El constructor matriz_segura_din.
  matriz_segura_din();

  // El constructor de copias matriz_segura_din.
  matriz_segura_din(const matriz_segura_din &obj);

  // Libera la memoria asignada cuando un objeto de
  // matriz_segura_din sale del ámbito.
  ~matriz_segura_din() {
     delete [] aptm;
  }

  // Sobrecarga la asignación.
  matriz_segura_din &operator=(const matriz_segura_din<T,longi> &op_der);

  // Usa el operador de subíndice para acceder a elementos en
  // la matriz segura.
  T &operator[](int i);

  // Devuelve el tamaño de la matriz.
```

```
  int getlen() { return longitud; }
};

// Esto es un constructor de matriz_segura_din.
template <class T, int longi>
  matriz_segura_din<T, longi>::matriz_segura_din() {

  try {
    // Asigna la matriz.
    aptm = new T[longi];
  } catch(bad_alloc ba) {
    cout << "No puede asignar la matriz.\n";
    // Tome aquí la acción apropiada. Esto es sólo
    // una respuesta de marcador de posición.
    exit(1);
  }

  // Inicializa los elementos de la matriz a su valor predeterminado.
  for(int i=0; i < longi; ++i) aptm[i] = T();

  longitud = longi;
}


// Esto es el constructor de copias de matriz_segura_din.
template <class T, int longi>
  matriz_segura_din<T, longi>::matriz_segura_din(const matriz_segura_din &obj) {

  cout << "Usando el constructor de copias de matriz_segura_din para hacer una
copia.\n";

  try {
    // Asigna una matriz del mismo tamaño que la
    // usada por obj.
    aptm = new T[obj.longitud];
  } catch(bad_alloc ba) {
    // Tome aquí la acción apropiada. Esto es sólo
    // una respuesta de marcador de posición.
    cout << "No puede asignar una matriz.\n";
    exit(1);
  }
  longitud = obj.longitud;

  // Copia el contenido de la matriz.
  for(int i=0; i < longitud; ++i)
    aptm[i] = obj.aptm[i];
}

// Sobrecarga de asignación para que se haga una copia de la
// matriz. La copia se almacena en una memoria asignada que está
// separada del operando del lado derecho.
//
template<class T, int longi> matriz_segura_din<T, longi> &
  matriz_segura_din<T, longi>::operator=(const matriz_segura_din<T, longi> &op_
der) {
```

```
  cout << "Asignando un objeto de matriz_segura_din a otro.\n";

  // Si es necesario, libere la memoria usada por el objeto.
  if(aptm && (longitud != op_der.longitud)) {

    // Elimine la memoria previamente asignada.
    delete aptm;

    try {
      // Asigna una matriz del mismo tamaño que el usado
      // por op_der.
      aptm = new T[op_der.longitud];
    } catch(bad_alloc ba) {
      // Tome aquí la acción apropiada. Esto es sólo una
      // respuesta de marcador de posición.
      cout << "No se puede asignar la matriz.\n";
      exit(1);
    }
  }

  longitud = op_der.longitud;

  // Copia el contenido de la matriz.
  for(int i=0; i < longitud; ++i)
    aptm[i] = op_der.aptm[i];
    return *this;
}

// Proporciona revisión de rango para matriz_segura_din al
// sobrecargar el operador []. Observe que se devuelve una
// referencia. Esto deja que se asigne un valor a un elemento
// de la matriz.
template <class T, int longi> T &matriz_segura_din<T, longi>::operator[](int i)
{
  if(i < 0 || i > longitud) {
    // Tome aquí la acción apropiada. Esto es sólo
    // una respuesta de marcador de posición.
    cout << "\nEl valor de \u00a1ndice de " << i << " est\u00a0 fuera del l\u00a-
mite.\n";
    exit(1);
  }
  return aptm[i];
}

// Una función simple para fines de demostración.
// Cuando se le llama, el constructor de copias se
// usará para crear una copia del argumento pasado a x.
template <class T, int longi>
  matriz_segura_din<T, longi> f(matriz_segura_din<T, longi> x) {

  cout << "f() est\u00a0 devolviendo una copia de x.\n";
  return x;
}
```

```
// Esto es una clase simple usada para demostrar una matriz de objetos.
// Observe que el constructor predeterminado da a x el valor -1.
class miclase {
public:
  int x;
  miclase(int i) { x = i; };
  miclase() { x = -1; }
};

int main()
{

  // Usa la matriz de enteros.
  matriz_segura_din<int, 5> mz_int;

  for(int i=0; i < mz_int.getlen(); ++i) mz_int[i] = i;
  cout << "Contenido de mz_int: ";
  for(int i=0; i < mz_int.getlen(); ++i) cout << mz_int[i] << " ";
  cout << "\n\n";

  // Para generar un desbordamiento de límites, quite las líneas de
  // comentario de la siguiente línea:
//  mz_int[19] = 10;

  // Para que se quede corto ante un límite, quite las líneas de
  // comentario de la siguiente línea:
//  mz_int[-2] = 10;

  // Crea una copia de mz_int. Esto invocará el constructor de copias de
  // matriz_segura_din.
  cout << "Crea mz_int2 y lo inicializa con mz_int. Esto da como resultado\n"
       << "que se llame a un constructor de copias de matriz_segura_din.\n\n";
  matriz_segura_din<int, 5> mz_int2 = mz_int;
  cout << "Contenido de mz_int2: ";
  for(int i=0; i < mz_int2.getlen(); ++i)  cout << mz_int2[i] << " ";
  cout << "\n\n";

  // Crea otra matriz segura para enteros, pero no le asigna
  // valores. Esto significa que sus elementos contendrán
  // sus valores predeterminados.
  cout << "Crea mz_int3.\n";
  matriz_segura_din<int, 5> mz_int3;

  cout << "Contenido original de mz_int3: ";
  for(int i=0; i < mz_int3.getlen(); ++i)  cout << mz_int3[i] << " ";
  cout <<"\n\n";

  // Ahora, pasa mz_int3 a f() y asigna el resultado a mz_int:
  cout << "Ahora, se ejecutar\u00a0 esta l\u00a1nea: mz_int3 = f(mz_int);\n"
       << "Esto da lugar a la siguiente secuencia de eventos:\n"
       << "   1. Se llama al constructor de copias de matriz_segura_din\n"
       << "      para copiar mz_int que se pasa al par\u00a0metro x de f().\n"
```

```
        << "   2. Se llama de nuevo al constructor de copias cuando se\n"
        << "      hace una copia para el valor devuelto de f().\n"
        << "   3. Se llama al operador de asignaci\u00a2n sobrecargado\n"
        << "      para asignar el resultado de f() a mz_int3.\n\n";
  mz_int3 = f(mz_int);

  cout << "Contenido de mz_int3 tras recibir el valor de f(mz_int): ";
  for(int i=0; i < mz_int3.getlen(); ++i)  cout << mz_int3[i] << " ";
  cout << "\n\n";

  cout << "Por supuesto, matriz_segura_din adem\u00a0s funciona con tipos de cla-
se.\n";
  matriz_segura_din<miclase, 3> mz_mc;
  cout << "Contenido original de mz_mc: ";
  for(int i=0; i < mz_mc.getlen(); ++i) cout << mz_mc[i].x << " ";
  cout << endl;
  mz_mc[0].x = 9;
  mz_mc[1].x = 8;
  mz_mc[2].x = 7;
  cout << "Valores en mz_mc tras establecerlos: ";
  for(int i=0; i < mz_mc.getlen(); ++i) cout << mz_mc[i].x << " ";
  cout << "\n\n";

  cout << "Ahora, se crea mz_mc2 y luego se ejecuta esta instrucci\u00a2n:\n"
       << "   mz_mc2 = f(mz_mc);\n\n";
  matriz_segura_din<miclase, 3> mz_mc2;
  mz_mc2 = f(mz_mc);
  cout << "Contenido de mz_mc2 tras recibir f(mz_mc): ";
  for(int i=0; i < mz_mc2.getlen(); ++i) cout << mz_mc2[i].x << " ";
  cout << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Contenido de mz_int: 0 1 2 3 4

Crea mz_int2 y lo inicializa con mz_int. Esto da como resultado
que se llame a un constructor de copias de matriz_segura_din.

Usando el constructor de copias de matriz_segura_din para hacer una copia.
Contenido de mz_int2: 0 1 2 3 4

Crea mz_int3.
Contenido original de mz_int3: 0 0 0 0 0

Ahora, se ejecutará esta línea: mz_int3 = f(mz_int);
Esto da lugar a la siguiente secuencia de eventos:
   1. Se llama al constructor de copias de matriz_segura_din
      para copiar mz_int que se pasa al parámetro x de f().
   2. Se llama de nuevo al constructor de copias cuando se
      hace una copia para el valor devuelto de f().
   3. Se llama al operador de asignación sobrecargado
      para asignar el resultado de f() a mz_int3.
```

```
Usando el constructor de copias de matriz_segura_din para hacer una copia.
f() está devolviendo una copia de x.
Usando el constructor de copias de matriz_segura_din para hacer una copia.
Asignando un objeto de matriz_segura_din a otro.
Contenido de mz_int3 tras recibir el valor de f(mz_int): 0 1 2 3 4

Por supuesto, matriz_segura_din además funciona con tipos de clase.
Contenido original de mz_mc: -1 -1 -1
Valores en mz_mc tras establecerlos: 9 8 7

Ahora, se crea mz_mc2 y luego se ejecuta esta instrucción:
   mz_mc2 = f(mz_mc);

Usando el constructor de copias de matriz_segura_din para hacer una copia.
f() está devolviendo una copia de x.
Usando el constructor de copias de matriz_segura_din para hacer una copia.
Asignando un objeto de matriz_segura_din a otro.
Contenido de mz_mc2 tras recibir f(mz_mc): 9 8 7
```

Opciones

Como se explicó en el análisis, la forma más común de constructor de copias sólo tiene un parámetro que es una referencia a un objeto de la clase para la que está definido el constructor de copias. Sin embargo, es permisible para un constructor de copias que tenga parámetros adicionales, siempre y cuando tengan argumentos predeterminados. Por ejemplo, suponiendo la clase **matriz_segura_din**, la siguiente declaración especifica un constructor de copias válido:

```
matriz_segura_din(const matriz_segura_din &obj, int num = -1);
```

Aquí, la opción predeterminada del parámetro **num** es –1. Podría usar este constructor para permitir que sólo los primeros **num** elementos de la nueva **matriz_segura_din** se inicialicen con los primeros **num** elementos de **obj**. Los elementos restantes pueden darse a un valor predeterminado. Cuando **num** es –1, toda la matriz se inicializa con **obj**. Esta versión del constructor de copias podría escribirse así:

```
// Si num no es -1, inicializa los primeros num elementos de una matriz segura
// usando el valor de obj. Los otros elementos obtienen valores predeterminados.
// De otra manera, inicializa toda la matriz con los elementos de obj.
template <class T, int longi>
  matriz_segura_din<T, longi>::matriz_segura_din(const matriz_segura_din &obj,
                                                  int num) {

  cout << "Usando el constructor de copias de matriz_segura_din para hacer una
copia.\n";

  try {
    // Asigna una matriz del mismo tamaño que la
    // usada por obj.
    aptm = new T[obj.longitud];
  } catch(bad_alloc ba) {
    // Tome aquí la acción apropiada. Esto es sólo
    // una respuesta de marcador de posición.
    cout << "No puede asignar una matriz.\n";
```

```
    exit(1);
  }
  longitud = obj.longitud;

  // Copia el contenido de obj, hasta el número pasado mediante num.
  // Si num es -1, entonces se copian todos los valores.
  if(num == -1) num = obj.longitud;

  for(int i=0; i < num; ++i)
    aptm[i] = obj.aptm[i];

  // Inicializa cualquier elemento restante con su valor predeterminado.
  for(int i=num; i < longitud; ++i)
    aptm[i] = T();
}
```

Podría utilizar este constructor como se muestra aquí:

```
matriz_segura_din<int, 5> mz_int2 (mz_int, 3);
```

Aquí, los primeros tres elementos de **mz_int** se usan para inicializar los primeros tres elementos de **mz_int2**. A los elementos restantes se les da un valor predeterminado, que para enteros es cero.

Como se explicó en el análisis (y se demostró con la clase **matriz_segura_din** en el Ejemplo adicional), si necesita implementar un constructor de copias, a menudo también necesita sobrecargar el operador de asignaciones. La razón es que el mismo problema que necesita usar el constructor de copias también estará presente durante la asignación. Es importante no subestimar la asignación.

Determine un tipo de objeto en tiempo de ejecución

Componentes clave		
Encabezados	**Clases**	**Funciones**
<typeinfo>	type_info	bool operator==(const type_info &*ob*) const bool operator!=(const type_info &*ob*) const bool before(const type_info &*ob*) const const char *name() const

En lenguajes polimórficos como C++, puede haber situaciones en que el tipo de un objeto es desconocido en tiempo de compilación debido a que la naturaleza precisa de ese objeto no se determina sino hasta que el programa se ejecuta. Recuerde que C++ implementa polimorfismo mediante el uso de jerarquías de clase, funciones virtuales y apuntadores a clases base. Debido a que un apuntador a clase base puede usarse para señalar un objeto de la clase base de *cualquier objeto derivado de esa base*, no siempre es posible saber de antemano cuál tipo de objeto será señalado por un

apuntador a la base. Esta determinación debe hacerse en tiempo de ejecución, empleando información de tipo en tiempo de ejecución (RTTI, RunTime Type Information). La característica clave que permite esto es el operador **typeid**. Para algunos lectores, RTTI y **typeid** son características bien comprendidas, pero para otros son la fuente de muchas preguntas. Por esto, en esta solución se describen las técnicas básicas de RTTI.

Paso a paso

Para identificar el tipo de un objeto en tiempo de ejecución se requieren los pasos siguientes:

1. Para obtener el tipo de un objeto, use **typeid(*objeto*)**. Devuelve una instancia de **type_info** que describe el tipo de *objeto*.
2. Para obtener una instancia de **type_info** para un tipo específico, use **typeid(*tipo*)**. Devuelve un objeto de **type_info** que representa *tipo*.

Análisis

Para obtener un tipo de objeto, use el operador **typeid**. Tiene dos formas. La primera se usa para determinar el tipo de un objeto. Se muestra aquí:

typeid(*objeto*)

Aquí, *objeto* es una expresión que describe el objeto cuyo tipo estará obteniendo. Éste puede ser el propio objeto, un apuntador al que se quita la referencia, o una referencia al objeto. **typeid** devuelve una referencia a un objeto **const** de tipo **type_info** que describe el tipo de *objeto*. La clase **type_info** está declarada en el encabezado **<typeinfo>**. Por tanto, debe incluirlo cuando usa **typeid**.

La clase **type_info** define los siguientes miembros públicos:

const char *name() const

bool operator==(const type_info &*ob*) const

bool operator!=(const type_info &*ob*) const

bool before(const type_info &*ob*) const

La función **name()** devuelve un apuntador al nombre del tipo, representado como una cadena terminada en un carácter nulo. Por ejemplo, suponiendo algún objeto llamado **obj**, la siguiente instrucción despliega el nombre de tipo del objeto:

```
cout << typeid(obj).name();
```

Los **==** y **!=** sobrecargados funcionan para la comparación de tipos. La función **before()** devuelve true si el objeto que invoca está antes del objeto usado como un parámetro en orden de intercalación. (Esta función no tiene nada que hacer con la herencia o las jerarquías de clase.)

La segunda forma de **typeid** toma un nombre de tipo como su argumento. Aquí se muestra:

typeid(*nombre-tipo*)

Aquí *nombre-tipo* especifica un nombre de tipo válido, como **int**, **string**, **vector**, etc. Por ejemplo, la siguiente expresión es perfectamente aceptable:

```
typeid(int).name()
```

Aquí, **typeid** devuelve el objeto **type_info** que describe **int**. El principal uso de esta forma de **typeid** consiste en comparar un tipo desconocido con uno conocido. Por ejemplo,

```
if(typeid(int) == typeid(*apt)) ...
```

Si **apt** señala a un **int**, entonces será correcta la instrucción **if**.

El uso más importante de **typeid** se presenta cuando se aplica mediante un apuntador de una clase base polimórfica. En este caso, automáticamente devolverá el tipo del objeto al que se está señalando. Recuerde que un apuntador de clase base puede señalar a objetos de la clase base o a un objeto de cualquier clase derivada de esa base. En todos los casos, **typeid** devuelve el tipo más derivado. Por tanto, si el apuntador señala a un objeto de clase base, entonces se devuelve el tipo de clase base. Si el apuntador señala a un objeto de clase derivada, se devuelve el tipo de clase derivada. Por tanto, **typeid** le permite determinar en tiempo de ejecución el tipo del objeto al que se está señalando mediante un apuntador a clase base.

Las referencias a un objeto de una jerarquía de clase polimórfica funcionan igual que los apuntadores. Cuando se aplica **typeid** a una referencia a un objeto de una clase polimórfica, devolverá el tipo de objeto al que se está haciendo referencia, que puede ser un tipo derivado. La circunstancia en que hará uso de esta característica con más frecuencia es cuando los objetos se pasan a funciones por referencia.

Si aplica **typeid** a un apuntador o referencia a un objeto de una jerarquía de clases no polimórfica, entonces se obtiene el tipo base del apuntador. Es decir, no se hace una determinación de lo que señala el apuntador.

Ejemplo

En el siguiente programa se demuestra el operador **typeid**. Crea una clase abstracta llamada **figura_dos_d** que define la dimensión de un objeto bidimensional, como un círculo o un triángulo. También especifica una pura función virtual llamada **area()**, que debe implementarse mediante una clase derivada para que devuelva el área de una forma. El programa crea tres subclases de **figura_dos_d**: **rectángulo**, **triángulo** y **círculo**.

El programa también define las funciones **trazador()** y **mismaforma()**. La función **original()** crea una instancia de una subclase de **figura_dos_d**, que será un **círculo**, **triángulo** o **rectángulo**, y devuelve un apuntador de **figura_dos_d** a él. El tipo específico de objeto creado se determina mediante la salida de una llamada a **rand()**, el generador de números aleatorios de C++. Por tanto, no hay manera de saber por anticipado qué tipo de objeto se generará. El programa crea seis objetos. Debido a que puede generarse cualquier tipo de figura mediante una llamada a **trazador()**, el programa depende de **typeid** para determinar qué tipo de objetos se han creado en realidad.

La función **mismaforma()** compara dos objetos de **figura_dos_d**. Los objetos son los mismos sólo si son del mismo tipo y tienen las mismas dimensiones. Utiliza **typeid** para confirmar que los objetos son del mismo tipo.

```
// Demuestra el id de tipo en tiempo de ejecución.

#include <iostream>
#include <cstdlib>

using namespace std;

// Una clase polimórfica que encapsula formas bidimensionales,
// como triángulos, rectángulos y círculos. Declara una
// función virtual llamada área(), cuyas clases derivadas
// se sobrecargan para calcular y devolver el área de una figura.
class figura_dos_d {
```

```
protected:
  double x, y;
public:
  figura_dos_d(double i, double j) {
    x = i;
    y = j;
  }

  double getx() { return x; }
  double gety() { return y; }

  virtual double area() = 0;
};

// Crea una subclase de figura_dos_d para triángulos.
class triangulo : public figura_dos_d {
  public:
    triangulo(double i, double j) : figura_dos_d(i, j) { }

    double area() {
      return x * 0.5 * y;
    }
};

// Crea una subclase de figura_dos_d para rectángulos.
class rectangulo : public figura_dos_d {
  public:
    rectangulo(double i, double j) : figura_dos_d(i, j) { }

    double area() {
      return x * y;
    }
};

// Crea una subclase of figura_dos_d para círculos.
class circulo : public figura_dos_d {
  public:
    circulo(double i, double j=0) : figura_dos_d(i, j) { }

    double area() {
      return 3.14 * x * x;
    }
} ;

// Un trazador de objetos derivados de figura_dos_d.
figura_dos_d *trazador() {
  static double i = (rand() % 100) / 3.0, j = (rand() % 100) / 3.0;

  i += rand() % 10;
  j += rand() % 12;

  cout << "Generando objeto.\n";

  switch(rand() % 3 ) {
```

```
    case 0: return new circulo(i);
    case 1: return new triangulo(i, j);
    case 2: return new rectangulo(i, j);
  }

  return 0;
}

// Compara la igualdad de dos figuras. Esto significa que sus tipos
// y dimensiones deben ser iguales.
bool mismaforma(figura_dos_d *alfa, figura_dos_d *beta) {

  cout << "Comparando un objeto de " << typeid(*alfa).name()
       << " con un objeto de " << typeid(*beta).name()
       << "object\n";

  if(typeid(*alfa) != typeid(*beta)) return false;

  if(alfa->getx() != beta->getx() &&
     alfa->gety() != beta->gety())  return false;

  return true;
}

int main()
{
  // Crea un apuntador a clase base a figura_dos_d.
  figura_dos_d *a;

  // Genera objetos de figura_dos_d.
  for(int i=0; i < 6; i++) {
    // Genera un objeto.
    a = trazador();

    // Despliega el nombre del objeto.
    cout << "El objeto es " << typeid(*a).name() << endl;

    // Despliega su área.
    cout << "    El \u00a0rea es " << a->area() << endl;

    // Mantiene una cuenta de los tipos de objetos que se han generado.
    if(typeid(*a) == typeid(triangulo))
      cout << "    La base es " << a->getx() << " La altura es "
           << a->gety() << endl;

    else if(typeid(*a) == typeid(rectangulo))
      cout << "    El largo es " << a->getx() << " La altura es "
           << a->gety() << endl;

    else if(typeid(*a) == typeid(circulo))
      cout << "    El di\u00a0metro es " << a->getx() << endl;

    cout << endl;
  }
```

```
  cout << endl;

  // Crea algunos objetos para comparar.
  triangulo t(2, 3);
  triangulo t2(2, 3);
  triangulo t3(3, 2);
  rectangulo r(2, 3);

  // Compara dos objetos de figura_dos_d.
  if(mismaforma(&t, &t2))
    cout << "t y t2 son iguales.\n";

  if(!mismaforma(&t, &t3))
    cout << "t y t3 son diferentes.\n";

  if(!mismaforma(&t, &r))
    cout << "t y r son diferentes.\n";

  cout << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
Generando objeto.
El objeto es class rectangulo
    El área es 465.222
    El largo es 17.6667 La altura es 26.3333

Generando objeto.
El objeto es class circulo
    El área es 1474.06
    El diámetro es 21.6667

Generando objeto.
El objeto es class rectangulo
    El área es 954.556
    El largo es 23.6667 La altura es 40.3333

Generando objeto.
El objeto es class circulo
    El área es 2580.38
    El diámetro es 28.6667

Generando objeto.
El objeto es class triangulo
    El área es 776.278
    La base es 29.6667 La altura es 52.3333

Generando objeto.
El objeto es class circulo
    El área es 3148.72
    El diámetro es 31.6667
```

```
Comparando un objeto de class triangulo con un objeto de class triangulo
t y t2 son iguales.
Comparando un objeto de class triangulo con un objeto de class triangulo
t y t3 son diferentes.
Comparando un objeto de class triangulo con un objeto de class rectangulo
t y r son diferentes.
```

Opciones

El operador **typeid** puede aplicarse a clases de plantilla. El tipo de un objeto que es una instancia de una clase de plantilla está determinado, en parte, por los datos usados para sus parámetros de tipo cuando se crea una instancia del objeto. Dos instancias de la misma clase de plantilla que se crean usando datos diferentes son, por tanto, tipos diferentes. Por ejemplo, suponga la clase de plantilla **miclase**, que se muestra aquí:

```
template <class T> class miclase {
  // ...
};
```

La siguiente secuencia:

```
miclase<int> mc_int;
miclase<double> mc_dbl;

cout << "El tipo de mc_int es " << typeid(mc_int).name() << endl
     << "El tipo de mc_dbl es " << typeid(mc_dbl).name() << endl

if(typeid(mc_int) != typeid(mc_dbl))
  cout << "Los dos objetos son de tipo diferente";
```

produce la siguiente salida:

```
El tipo de mc_int es miclase<int>
El tipo de mc_dbl es miclase<double>
Los dos objetos son de tipo diferente
```

Como puede ver, aunque **mc_int** y **mc_dbl** son objetos de **miclase**, sus tipos difieren porque se usan diferentes plantillas de argumentos.

Use números complejos

Componentes clave		
Encabezados	**Clases**	**Funciones**
<complex>	complex	T imag() const T real() const

Una característica en ocasiones subestimada de C++ es el soporte a números complejos. Un número complejo contiene dos componentes: una parte real y una imaginaria. Esta última especifica un múltiplo de *i*, que es la raíz cuadrada de –1. Por tanto, un número complejo suele representarse de esta forma:

$a + bi$

donde *a* especifica la parte real y *b* la imaginaria. En C++, los números complejos tienen soporte con la clase **complex**. En esta solución se muestran las técnicas básicas para usarla.

Paso a paso

Para usar números complejos se requieren estos pasos:

1. Cree uno o más objetos de **complex**. La clase **complex** es genérica, y usted debe especificar el tipo de los componentes. Por lo general, esto será un tipo de punto flotante, como **double**.
2. Realice operaciones con objetos de **complex** al usar operadores sobrecargados. Todos los operadores aritméticos están definidos por **complex**.
3. Obtenga el componente real de una instancia de **complex** al llamar a **real()**.
4. Obtenga el componente imaginario de una instancia de **complex** al llamar a **imag()**.

Análisis

La especificación de plantilla para **complex** se muestra a continuación:

```
template <class T> class complex
```

Aquí, **T** especifica el tipo usado para representar los componentes de un número complejo. Hay tres especializaciones predefinidas de **complex**:

```
class complex<float>
class complex<double>
class complex<long double>
```

No está definida la especificación de algún otro argumento de tipo.

La clase **complex** tiene los siguientes constructores:

```
complex(const T &real = T(), const T &imaginario = T())
complex(const complex &ob)
template <class T1> complex(const complex<T1> &ob);
```

El primero construye un objeto de **complex** con un componente real de *real* y uno imaginario de *imaginario*. El valor predeterminado de estos valores es cero, si no está especificado. El segundo crea una copia de *ob*. El tercero crea un objeto de **complex** a partir de *ob*.

Las siguientes operaciones están definidas para objetos de **complex**:

+	–	*	/
–=	+=	/=	*=
=	==	!=	

Los operadores sin asignación se sobrecargan de tres maneras: una vez para operadores que requieren un objeto de **complex** a la izquierda y un objeto escalar a la derecha, una vez más para operaciones que requieren un objeto escalar a la izquierda y uno de **complex** a la derecha, y finalmente para operaciones que requieren dos objetos de **complex**. Por ejemplo, los siguientes tipos de operaciones de suma están permitidos:

ob_complex + escalar

escalar + ob_complex

ob_complex + ob_complex

Están definidas dos funciones miembro para **complex**: **real()** e **imag()**. Aquí se muestran:

T real() const

T imag() constructores

La función **real()** devuelve el componente real del objeto que invoca, e **imag()** devuelve el componente imaginario.

El encabezado **<complex>** también define versiones de **complex** de las funciones matemáticas estándar, como **abs()**, **sin()**, **cos()** y **pow()**.

Ejemplo

He aquí un programa de ejemplo que demuestra **complex**:

```
// Demuestra la clase complex.

#include <iostream>
#include <complex>

using namespace std;

int main()
{
  complex<double> cmpx1(1, 0);
  complex<double> cmpx2(1, 1);

  cout << "cmpx1: " << cmpx1 << endl << "cmpx2: " << cmpx2 << endl;

  // Suma dos números complejos.
  cout << "cmpx1 + cmpx2: " << cmpx1 + cmpx2 << endl;

  // Multiplica dos números complejos.
  cout << "cmpx1 * cmpx2: " << cmpx1 * cmpx2 << endl;
```

```
  // Suma un número escalar a uno complejo.
  cmpx1 +=  2.0;
  cout << "cmpx1 += 2.0: " << cmpx1 << endl;

  // Encuentra el seno de cmpx2.
  cout << "sin(cmpx2): " << sin(cmpx2) << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
cmpx1: (1,0)
cmpx2: (1,1)
cmpx1 + cmpx2: (2,1)
cmpx1 * cmpx2: (1,1)
cmpx1 += 2.0: (3,0)
sin(cmpx2): (1.29846,0.634964)
```

Opciones

Para el caso de programadores que se concentran en cálculos numéricos, C++ provee más soporte del que se podría imaginar. Además de **complex**, C++ incluye la clase **valarray** que da soporte a operaciones de matrices numéricas. También proporciona dos clases de utilería llamadas **slice** y **gslice**, que encapsulan una parte (es decir, una porción o "rebanada") de una matriz. Estas clases requieren el encabezado **<valarray>**. En el encabezado **<numeric>** están definidos cuatro algoritmos numéricos llamados **accumulate()**, **adjacent_difference()**, **inner_product()** y **partial_sum()**. Todos tienen algún interés para el programador.

Use auto_ptr

Componentes clave		
Encabezados	**Clases**	**Funciones**
<memory>	auto_ptr	T *get() const throw() T*release() throw() Void reset(X **ptr* = 0) throw ()

C++ incluye una clase llamada **auto_ptr** que se diseñó para simplificar la administración de memoria asignada dinámicamente. Como muchos lectores sabrán, uno de los aspectos del uso de la asignación dinámica que provoca más problemas es la prevención de las fugas de memoria. Una manera en que ocurre una fuga de memoria es cuando se asigna ésta, pero nunca se libera. La clase **auto_ptr** representa un intento por prevenir esta situación. En esta solución se describe su uso.

Paso a paso

Para usar **auto_ptr** se necesitan estos pasos:

1. Cree un **auto_ptr**, especificando el tipo de base del apuntador.
2. Asigne memoria usando **new**, y asigne un apt a la memoria al **auto_ptr** creado en el paso 1.
3. Use el **auto_ptr** como un apuntador normal. Sin embargo, no libere la memoria a la que apunta **auto_ptr**. En otras palabras, no use **delete** para liberar la memoria.
4. Cuando se destruye el **auto_ptr**, como cuando sale del ámbito, se libera automáticamente la memoria a la que señala **auto_ptr**.
5. Puede obtener el apuntador contenido por un **auto_ptr** al llamar a **get()**.
6. Puede establecer el apuntador de **auto_ptr** al llamar a **reset()**.
7. Puede liberar la propiedad de **auto_ptr** del apuntador al llamar a **release()**.

Análisis

Un **auto_ptr** es un apuntador que posee el objeto al que señala. La propiedad de este objeto puede transferirse a otro **auto_ptr**, pero algún **auto_ptr** siempre posee el objeto. Por ejemplo, cuando un objeto de **auto_ptr** se asigna a otro, sólo el destino de la asignación será su propietario. Cuando se destruye un **auto_ptr**, como cuando sale del ámbito, el objeto al que señala **auto_ptr** se libera automáticamente. Debido a que sólo un **auto_ptr** poseerá (contendrá un apuntador a) cualquier objeto determinado en cualquier momento dado, el objeto sólo se liberará una vez, cuando se destruye el **auto_ptr** que tiene la propiedad. Cualquier otro **auto_ptr** que previamente haya tenido la propiedad no entrará en acción. El mecanismo asegura que los objetos asignados dinámicamente se liberen apropiadamente en todas las circunstancias. Entre otros beneficios de este método se encuentra el de que los objetos asignados dinámicamente pueden liberarse de manera automática sin que ocurra una excepción.

La especificación de plantilla para **auto_ptr** se muestra a continuación:

```
template <class T> class auto_ptr
```

Aquí, **T** especifica el tipo de apuntador almacenado por **auto_ptr**.

He aquí el constructor para **auto_ptr**:

```
explicit auto_ptr(T *apt = 0) throw()
auto_ptr(auto_ptr &ob) throw()
template <class T2> auto_ptr(auto_ptr<T2> &ob) throw()
```

El primer constructor crea un **auto_ptr** al objeto especificado por *apt*. El segundo crea una copia de **auto_ptr** especificada por *ob* y trasfiere la propiedad al nuevo objeto. El tercero convierte &*ob* al tipo **T** * (si es posible) y transfiere la propiedad.

La clase **auto_ptr** define los operadores **=**, ***** y **–>**. También define estas tres funciones:

T *get() const throw()

T *release() throw()

void reset(X **apt* = 0) throw()

La función **get()** devuelve un apuntador al objeto almacenado. La función **release()** elimina la propiedad del objeto almacenado del **auto_ptr** que invoca y devuelve un apuntador al objeto. Después de una llamada a **release()**, el objeto al que se apunta no se destruye automáticamente cuando el objeto **auto_ptr** sale del ámbito. La función **reset()** llama a **delete** en el apuntador contenido por **auto_ptr** (a menos que sea igual a *apt*) y luego establece el apuntador a *apt*.

Ejemplo

He aquí un programa corto que demuestra el uso de **auto_ptr**. Crea una clase llamada **X** que almacena un valor entero. Dentro de **main()**, se crea un objeto **X** y se asigna a un **auto_ptr**. Observe cómo se tiene acceso a los miembros de **X** mediante el **auto_ptr**, usando el operador de apuntador normal **–>**. Además, observe cómo uno y sólo uno de los **auto_ptr** posee el apuntador al objeto de **X** en un momento determinado. Ésta es la razón por la que sólo un objeto de **X** se destruye cuando termina el programa.

```
// Demuestra un auto_ptr.

#include <iostream>
#include <memory>

using namespace std;

class X {
public:
  int v;

  X(int j) {
    v = j;
    cout << "Construyendo X(" << v <<")\n";
  }

  ~X() { cout << "Destruyendo X(" << v <<")\n"; }


  void f() { cout << "Dentro de f()\n"; }
};

int main()
{
  auto_ptr<X> a1(new X(3)), a2;

  cout << "a1 apunta a un X con el valor " << a1->v
       << "\n\n";

  // Transfiere la propiedad a a2.
  cout << "Asignando a1 a a2.\n";
```

```
  a2 = a1;
  cout << "Ahora, a2 apunta a un X con el valor " << a2->v
       << endl;
  if(!a1.get()) cout << "El apuntador de a1 ahora es null.\n\n";

  // Puede llamar a una función mediante un auto_ptr.
  cout << "Llama a f() mediante a2: ";
  a2->f();
  cout << endl;

  // Asigna al apuntador encapsulado por un auto_ptr a
  // un apuntador normal.
  cout << "Obtiene el apuntador almacenado en a2 y lo asigna al \n"
       << "apuntador normal llamado apt.\n";
  X *apt = a2.get();
  cout << "apt apunta a un X con el valor " << apt->v
       << "\n\n";

  return 0;

  // En este momento, el objeto asignado se libera y
  // se llama a su destructor. Aunque hay dos objetos
  // de auto_ptr, sólo uno posee el apuntador. Por
  // tanto, sólo se destruye un objeto de X.
}
```

Aquí se muestra la salida producida por este programa:

```
Construyendo X(3)
a1 apunta a un X con el valor 3

Asignando a1 a a2.
Ahora, a2 apunta a un X con el valor 3
El apuntador de a1 ahora es null.

Llama a f() mediante a2: Dentro de f()

Obtiene el apuntador almacenado en a2 y lo asigna al
apuntador normal llamado apt.
apt apunta a un X con el valor 3

Destruyendo X(3)
```

Opciones

Aunque **auto_ptr** es útil, no evita todos los problemas relacionados con los apuntadores. Por ejemplo, aún es posible operar por accidente sobre un apuntador nulo. Sin embargo, puede usar un **auto_ptr** como base para su propio tipo personalizado de "apuntador seguro". Para experimentar con este concepto, pruebe el uso de **auto_ptr** para el miembro **apt** de la clase **apt_seguro** mostrada en el Ejemplo adicional de *Sobrecargue el operador –>*.

Otra cosa que **auto_ptr** no proporciona es *recolección de basura*. Como casi todos los lectores saben, la recolección de basura es el esquema de administración de memoria en que la memoria se recicla automáticamente cuando ya no se usa en algún objeto. Aunque aspectos de **auto_ptr**

parecen relacionados con la recolección de basura, como el hecho de que la memoria asignada se libera automáticamente cuando el **auto_ptr** sale del ámbito, la recolección de basura depende de un mecanismo fundamentalmente diferente. En la actualidad, el C++ estándar no define una biblioteca de recolección de basura, pero es probable que la siguiente versión de C++ sí la incluya.

Un tema adicional: para pasar un **auto_ptr** a una función, recomiendo el uso de un parámetro de referencia. En el transcurso de los años, se han visto cambios importantes en la manera en que diferentes compiladores manejan el paso de un valor de **auto_ptr**. El paso de una referencia evita el problema.

Cree un constructor explícito

Componentes clave		
Encabezados	**Clases**	**Funciones**
	cualquier clase	explicit *constructor*(*tipo param*)

Para concluir este libro de C++, se ha elegido una de sus características más esotéricas: el constructor explícito. Con los años, el autor se ha preguntado varias veces acerca de esta característica, porque se usa con frecuencia en la biblioteca estándar de C++. Aunque no es difícil, resulta una característica especializada cuyo significado no se comprende universalmente. En esta solución se describe el objetivo de un constructor explícito y se muestra cómo crear uno.

Paso a paso

Para crear un constructor explícito se necesitan estos pasos:

1. Cree un constructor que tome un argumento.
2. Modifique el constructor con la palabra clave **explicit**.

Análisis

C++ define la palabra clave **explicit** para que maneje una condición especial de caso que ocurre con un constructor que requiere sólo un argumento. Para comprender el propósito de **explicit**, considere la siguiente clase:

```
class miclase {
  int val;
public:
  miclase(int x) { val = x; }
  int getval() { return val; }
};
```

Observe que el constructor de **miclase** tiene un parámetro. Esto significa que puede crear un objeto de **miclase** como éste:

```
miclase ob(4);
```

En esta declaración, el valor 4, que se especifica entre paréntesis después de **ob**, es un argumento pasado al parámetro **x** de **miclase()**. Este valor se usa después para inicializar **val**. Se trata de una forma común de inicialización, y se usa ampliamente en este libro. Sin embargo, hay una opción, como se muestra en la siguiente instrucción, que también inicializa **val** en 4:

```
miclase ob = 4; // se convierte automáticamente en miclase(4)
```

Como lo sugiere el comentario, esta forma de inicialización se convierte automáticamente en una llamada al constructor de **miclase**, y 4 es el argumento. Es decir, el compilador maneja la instrucción anterior como si fuera ésta:

```
miclase ob(4);
```

En general, en cualquier momento en que tenga un constructor que requiera sólo un argumento, puede usar *ob*(*x*) u *ob* = *x* para inicializar un objeto. La razón es que cada vez que cree un constructor que requiera un argumento, está creando implícitamente una conversión del tipo de ese argumento al de la clase.

Si no quiere que ocurran conversiones implícitas, puede evitarlas al usar **explicit**. Este especificador sólo se aplica a constructores. Un constructor especificado como **explícito** sólo se usará cuando una inicialización use la sintaxis del constructor normal. No realizará ninguna conversión automáticamente. Por ejemplo, al declarar **explicit** el constructor de **miclase**, como se muestra aquí:

```
explicit miclase(int x) { val = x; }
```

no se proporcionará la conversión automática. Ahora, sólo se permitirán constructores de la forma

```
miclase ob(27);
```

Ya no se permitirá esta forma

```
miclase ob = 27; // ¡Ahora es un error!
```

Ejemplo

En el siguiente ejemplo se integran las piezas y se ilustra un constructor **explicit**. En primer lugar, he aquí un programa que ilustra la conversión automática que ocurre cuando un constructor no se modifica con **explicit**:

```
#include <iostream>

using namespace std;

class miclase {
  int val;
public:
  // El siguiente constructor NO es explícito.
  miclase(int x) { val = x; }
```

```
  int getval() { return val; }
};

int main()
{
  miclase ob(4); // Correcto
  cout << "val en ob: " << ob.getval() << endl;

  // La siguiente instrucción es correcta debido a la
  // conversión implícita de int a miclase.
  miclase ob2 = 19;
  cout << "val en ob2: " << ob2.getval() << endl;

  return 0;
}
```

Aquí se muestra la salida:

```
val en ob: 4
val en ob2: 19
```

Como puede ver, ambas formas de inicialización son permitidas, y ambas inicializan una instancia de **miclase**, como se esperaba.

La siguiente versión del programa agrega el modificador **explicit** al constructor de **miclase**:

```
#include <iostream>

using namespace std;

class miclase {
  int val;
public:
  // Ahora miclase(int) es explícito.
  explicit miclase(int x) { val = x; }

  int getval() { return val; }
};

int main()
{
  miclase ob(4); // Aún es correcto
  cout << "val en ob: " << ob.getval() << endl;

  // La siguiente instrucción es un error porque ya no está
  // permitida la conversión implícita de int a miclase.
  miclase ob2 = 19;  // ¡Error!
  cout << "val en ob2: " << ob.getval() << endl;

  return 0;
}
```

Después de hacer **miclase(int)** explícita, la instrucción

```
miclase ob2 = 19; //¡Error!
```

es ahora un error y no se compilará.

Opciones

El modificador **explicit** sólo se aplica a constructores que requieren un argumento. Sin embargo, esto no significa que el constructor deba tener un solo parámetro. Simplemente significa que cualquier parámetro después del primero debe tener argumentos predeterminados. Por ejemplo:

```
class miclase {
  int val;
  int otro_valor;
public:
  explicit miclase(int x, int y = 0) { val = x; otro_val = y; }
   // ...
};
```

Debido a que el valor predeterminado de **y** es 0, el uso de **explicit** aún es válido. Su uso evita la siguiente declaración:

```
miclase contador = 19; // no válida.
```

Si el constructor *no ha sido* declarado como **explicit**, la instrucción anterior se permitiría; **y** tendría el valor predeterminado de 0. Debido a **explicit**, es necesario invocar explícitamente al constructor, como en el ejemplo siguiente:

```
miclase contador(19);
```

Por supuesto, también puede especificar un segundo argumento:

```
miclase contador(19, 99);
```

Índice

Símbolos

A

D

E

F

G

N

Q

R

S